UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

DISEÑO Y PROGRAMACIÓN DE UN SISTEMA PARA LA DISPENSA AUTOMÁTICA DE PRECINTOS PARA CONTENEDORES MARITIMOS

TRABAJO FINAL DEL

Grado en Ingeniería Electrónica Industrial y Automática

REALIZADO POR

Daniel Díaz Lozano

TUTORIZADO POR

Carlos Ricolfe Viala

CURSO ACADÉMICO: 2020/2021

Índice

Contenido

Índice de imágenes

Objetivo

El objetivo del proyecto es programar un prototipo, que sirva de prueba de concepto de un sistema de dispensado de precintos, de tipo sello de perno de alta seguridad, como se definen en la norma ISO PAS 17712-2013. En este proyecto además se realizará la selección de los sensores y actuadores necesarios para su funcionamiento.

El concepto de dispensador que se trata de poner a prueba es el de un robot cartesiano con un sistema de visión 2D tradicional. Debe ser capaz de extraer de un recipiente, en el cual se han descargado de forma desordenada los precintos, un único precinto por activación, y entregarlo tras leer el código numérico impreso en el precinto. Además, debe estar diseñado con la idea de minimizar el mantenimiento necesario, en el caso de su instalación a largo plazo.

El proyecto se desarrolla con el instituto ai2, al ser un proyecto suyo, y no todo el desarrollo fue realizado por el alumno. En el ai2 hay servicios de diseño e impresión 3D, electrónica e informática con expertos en la materia que sirven de apoyo en esos temas para cualquier proyecto desarrollado en el ai2, y este proyecto no fue una excepción. Cuando esos servicios se utilicen, se nombrarán y los detalles no se darán en esta memoria.

Memoria

Antecedentes

El sistema portuario es clave para la economía mundial, y se encuentra en constante evolución para adaptarse a los cambios en los modelos de mercado, en busca de mayor rentabilidad y eficiencia.

En esta búsqueda de rentabilidad y eficiencia, la digitalización ha abierto la puerta al concepto de los puertos inteligentes, con el uso de tecnologías de información modernas para la planificación y administración, dentro y entre puertos. En esta línea de trabajo la Autoridad Portuaria de Valencia (APV) tiene en funcionamiento la plataforma electrónica ValenciaportPCS (Port Community Service).

En los últimos años la Autoridad Portuaria de Valencia (APV) ha tenido en desarrollo para su plataforma electrónica ValenciaportPCS (Port Community Service) una herramienta de gestión de precintos, que tiene como objetivo la reducción de tiempos y costes, de transportistas y navieras, y tener un mejor control del stock disponible.

Esta nueva herramienta fue presentada el 11 de diciembre de 2019, y ha sido desarrollada en conjunto con Cosco Shipping y la Asociación de Empresas de Logística y Transporte de Contenedores del puerto de Valencia (ELTC), y en la actualidad funciona con los contenedores de Cosco usando el ELTC como punto de entrega de los precintos.

Justificación

La nueva herramienta de gestión de precintos reduce las colas que se forman en los puntos de recogida de precintos, al agilizar las gestiones necesarias para asociar el número de precinto con el cargamento y el camión, pero no elimina la necesidad de desplazarse a los puntos de recogida, los cuales no pueden encontrarse demasiado cerca del punto de carga al tener que estar situados en unas oficinas.

Dentro del plan de la herramienta de gestión de precintos se proyecta la creación de una dispensadora de precintos conectada al PCS, que agilizaría aún más el proceso al terminar de automatizar la entrega de precintos a los transportistas, y al no depender de unas oficinas se podrían instalar mucha más cerca de los puntos de carga, reduciendo el tiempo que los transportistas deben dedicar a obtener los precintos para sus cargamentos.

Los puntos clave del concepto que se va a poner a prueba se justifican en este marco de necesidades de la siguiente manera:

El recipiente ha de contener los precintos descargados de forma desordenada, para que este proceso de automatización no traslade carga de trabajo al encargado de reponer, en vez de eliminarla del proceso.

El código impreso en el precinto se ha de extraer para tener la información necesaria para comunicarse con el PCS y que se puedan realizar las gestiones necesarias.

El tipo de robot y sistema de visión se escogen por ser opciones baratas en comparación con otras opciones disponibles, lo cual ayuda a mantener costes bajos.

Soluciones adoptadas

Método de extracción

El punto clave que más impacto supone, a la hora de tomar una decisión en este aspecto, es que los precintos hayan sido descargados de forma desordenada.

En caso de una descarga ordenada, como podría ser sobre unas guías, el precinto tiene una posición concreta, eliminando la necesidad de localizar el precinto entero o la parte relevante que se escoja para su extracción.

Con esto en mente, un sistema de visión artificial se vuelve necesario, y es algo en lo cual hemos impuesto limitaciones: debe ser barato.

Dejando de lado el sistema de visión, pues los detalles de este pueden variar en gran medida dependiendo de qué método se escoja para la extracción del precinto, lo siguiente que sería necesario plantear es cómo extraerlo del recipiente.

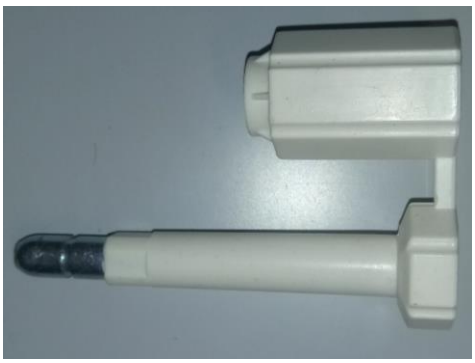


Imagen 1: Vista lateral del precinto.

El precinto, imagen 1, tiene un núcleo solido metálico, de material desconocido, pero por pruebas realizadas sabemos que es magnético, y en el caso de los precintos disponibles, las dos piezas se encuentran unidas por un trozo de plástico, algo que no siempre pasa en este tipo de precintos.

Se nos abren dos opciones principales a la hora de extraer el precinto: usar un imán, o coger el precinto, ya sea con una pinza o un gancho.

Ahora que sabemos que opciones se nos plantean para extraer el precinto, retomemos el punto del sistema de visión, y qué necesidades plantea cada opción de extracción en el sistema de visión:

- Imán: es necesario localizar un punto que en contacto con el imán quede magnetizado, la dirección del precinto no es relevante, pues no es necesario alinear nada.
- Pinza: es necesario localizar una de las dos piezas, la dirección en la que se encuentra es relevante para alinear la pinza con la pieza, y la posición Z
- Gancho: es necesario localizar un punto de fácil enganche, que en nuestro caso es el trozo de plástico que une ambas piezas del precinto, será necesaria la dirección de este, para calcular el ángulo de aproximación, y la posición Z.

Ahora que tenemos sobre la mesa las necesidades del sistema de visón en cada caso, hay que comprobar que suponen a la hora de diseñarlo:

En los tres casos es necesario localizar algo concreto, pero el imán es el único que no necesita también conocer la dirección de lo que debe encontrar el sistema de visión.

Tanto la pinza como el gancho necesitan localizar un objeto, no solo un punto, lo cual supone que es.

Imagen 2: caja de precintos

Añadiendo a la ecuación la imagen 2, una imagen extraída de la caja de precintos de la que se dispone para el proyecto, podemos empezar a tomar decisiones sobre el método de extracción.

Como se puede observar en la imagen 2, hay 3 cosas fáciles de ver a simple vista:

- El trozo de metal de la punta.
- Las sombras.
- Y el reflejo directo sobre el precinto.

Esta sería la información que se extraería de un sistema de visión 2D, sin procesar la imagen. Los objetos que tanto la pinza como el gancho necesitan localizar se mezclan con el resto al no tener un color distintivo, pero las sombras se podrían utilizar para generar un perímetro.

Existe un problema al hacer esto, y es que no siempre tienen la misma forma, en la imagen 2D, y que no hay manera de obtener la distancia Z al objeto, pues, aunque se puede hacer un cálculo comparando número de píxeles y tamaño real, las diferentes posiciones en las que se pueden encontrar hacen que el dato obtenido de número de pixeles no sea útil, lo cual imposibilita el uso del gancho y la pinza con visión 2D.

En caso de utilizar un sistema de visón 3D, estos datos se pueden obtener, pero el coste de una cámara 3D es mucho más elevado que el de una 2D, y los algoritmos a desarrollar son mucho más complejos.

En el caso de utilizar el imán, al no ser relevante que punto se escoja, el problema de la forma del objeto se ignora y se puede, simplemente, bajar el imán hasta hacer contacto para no ser necesario el dato de la posición Z.

La conclusión al problema del método de extracción será utilizar un imán.

Ahora falta comprobar los detalles de nuestro imán. Si va a ser un electroimán o un imán natural, y su potencia.

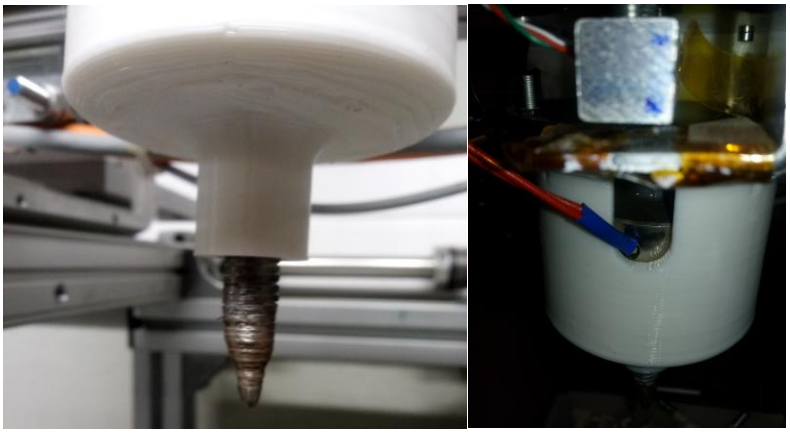
La primera decisión es simple, porque vamos a necesitar descargar el precinto, con lo que el imán necesita poder ser apagado, lo cual supone que un electroimán es necesario para el proyecto.

Para la segunda se van a hacer ensayos.

Se hicieron pruebas descendiendo el imán a mano sobre el recipiente, y se pudo observar un problema subyacente durante las mismas: los electroimanes son bastante grandes para este proceso.

Aunque los imanes más pequeños disponibles, que poseen forma cilíndrica y unos 3 cm de diámetro, son capaces de extraer un precinto al entrar en contacto con el trozo de metal visible, su tamaño seguía siendo demasiado grande y tendía a coger más precintos que se encontraban cerca.

La idea que se tuvo fue la de añadir una punta, de hierro, para que se magnetizase. Esto reduce la fuerza del campo magnético en el punto de contacto, pero al reducir la superficie de contacto se dejó de coger múltiples precintos, excepto cuando las partes de metal al descubierto se encontraban muy cerca. Haciendo pruebas con diferentes imanes y posibles puntas, teniendo en cuenta la perdida de fuerza al añadir la punta, se acabó utilizando un electroimán de 8 cm de diámetro, y un tornillo de 10 cm de largo y 1,5 cm de diámetro modificado con torno para hacer la punta y quitar la rosca.



a b

Imagen 3: Tornillo (a) y electroimán (b).

Sensorización de la recogida del precinto

Al elegir el imán, como el método de extracción a utilizar, hemos evitado la necesidad de obtener el dato de la posición Z del precinto a extraer, pero aún necesitamos confirmar que se recoge un precinto.

Necesitamos conseguir dos cosas con este sistema: sustituir la información que nos daría el dato Z del precinto, a la hora de bajar el imán en el recipiente, y confirmar que hay un precinto enganchado al levantar el imán.

En cuanto a bajar el imán hasta el precinto escogido para la extracción, vamos a necesitar sensorizar el contacto entre el imán y el precinto, al no tener información externa por el sistema de visión. Para esto podemos utilizar un sensor binario en algún punto del brazo, o un sensor no binario que mida la fuerza, colocado de manera similar a como pondríamos uno binario.

La elección de qué sensor utilizar la podemos relacionar con su utilidad a la hora de detectar si hay un precinto enganchado al levantar el imán, para reducir la cantidad de sistemas en el prototipo, y con ello su complejidad y precio.

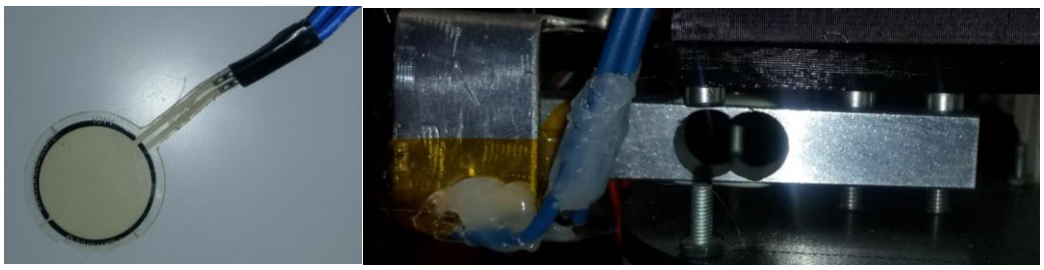
En el caso de un sensor binario, podemos detectar cuando no hay nada enganchado/tocando el imán, y cuando sí, con sus dos estados, así que podemos pensar en una manera de colocarlo para que sea funcional, pero hay un caso no contemplado: coger más de un precinto.

Este es un tercer estado, que no podemos medir con un sensor binario.

En el caso de un sensor de fuerza, podemos detectar una gran cantidad de estados al asociar un rango de mediciones a cada estado posible, como, por ejemplo, valores bajos para no contacto o no precinto cogido, valores medios para un precinto cogido o para cuando hay contacto al bajar, y valores altos para cuando hay más de un precinto cogido.

Con esto en mente vamos a escoger el sensor de fuerza no binario para sensorizar la recogida del precinto.

Falta elegir el tipo de sensor de fuerza. Se hicieron ensayos con un sensor de fuerza resistivo y una célula de carga.



a b

Imagen 4: sensor de fuerza (a) y célula de carga (b).

Ya que ninguno de los sensores tiene una ventaja clara de capacidades a simple vista, para decidir cuál utilizar, se realizarán pruebas sobre piezas prototipo.

Se comenzaron las pruebas con el sensor resistivo, ya que su menor tamaño sería más cómodo de implementar, pero enseguida se notó que tenía problemas de histéresis, así que se descartó.

A continuación, se realizaron las pruebas con la célula de carga, y ya que no hubo problemas, se decidió que se utilizaría este sensor.

Diseño del robot cartesiano

La caja de precintos disponible mide 23 cm de ancho por 23 cm de largo. Para los ejes X e Y serán necesarios actuadores con recorrido suficiente para cubrir la caja por completo. Se escogen para esta tarea dos ejes lineales de 30 cm de recorrido, con finales de carrera ajustables para limitar el movimiento al deseado, y que se controlan utilizando puerto serie. Para el eje Z se va a utilizar una guía con un tornillo sin fin accionado por un motor paso a paso.

Para el eje Z se solicitaron las piezas necesarias para poder unirlo al soporte que ya venía colocado en las guías lineales, cuando se desmontó de su proyecto original, al servicio de diseño e impresión 3D.

Diseño del cabezal del robot cartesiano

Con todo lo relacionado de forma directa con el robot cartesiano concretado, hay que plantear el diseño del cabezal que tiene que albergar todo.

El cabezal va al final de brazo del eje Z, y va a tener que poder albergar el electroimán, el tornillo que hace de punta para este, y tiene que, por cómo se instale en el brazo, servir para utilizar la célula de carga.

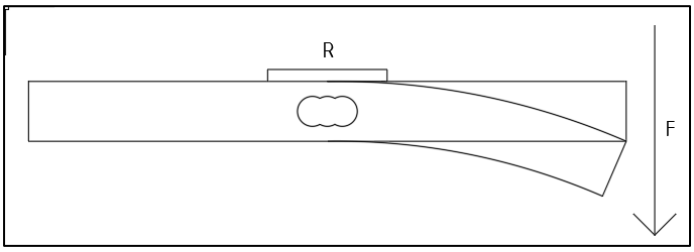


Imagen 5: diagrama de uso de la célula

Por como necesita ser usada la célula, el cabezal tendrá que ir enganchado a uno de sus extremos y tendremos que fijar el otro al brazo del eje Z, para que los cambios en la presión en la punta se transmitan directamente a deformación en la célula.

Debido al tamaño de la célula, para poder mantener centrada la punta en el brazo del eje Z y que se transmita la presión en la punta directamente en vertical sobre este, tendremos que desplazar bastante hacia el exterior el punto de enganche de la célula y el brazo.

Además, pondremos la célula en paralelo a la guía lineal del eje X, que es la que impone limites, y evitar que al bajar el brazo la célula pudiese chocar con esta.

En cuanto al diseño del cabezal, las condiciones son: que tenga hueco para atornillarlo a la célula y se sea capaz de albergar el electroimán y la punta. Este diseño se dejó al servicio de diseño e impresión 3D.

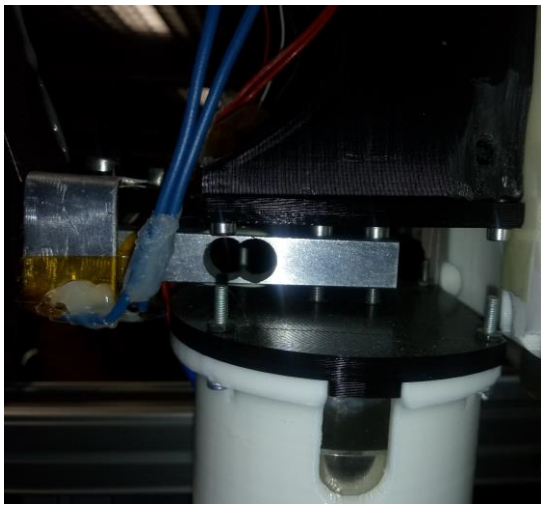


Imagen 6: enganche del cabezal.

Diseño del bloque de captura

Primero aclarar a qué nos referimos con bloque de captura: el bloque de captura será el punto del prototipo donde se descargue el precinto tras su extracción.

Aparte de esa función básica, vamos a añadir que sea capaz de capturar el código del precinto.

De solo necesitar que fuese el lugar donde descargar el precinto, sería suficiente con que se tratase de un recipiente con una tapa, por la cual el usuario fuese capaz de acceder para recogerlo, pero capturar el código impone condiciones a este recipiente.

- La primera condición será que necesita un sistema de visión para la captura del código.
- La segunda es que el precinto necesita caer en una posición concreta para poder utilizar un sistema de visión.

Más adelante, al concretar las dos primeras condiciones, aparecerá una tercera condición para el bloque de captura, pero se hablará de ella en el momento en el que surgió durante el desarrollo.

Imagen 7: precinto enseñando el código

Como se ve en la imagen 7, el precinto tiene un código de barras y un código numérico asociado en uno de sus lados. En este proyecto, como es un prototipo, vamos a leer el código numérico porque es fácil de comprobar a mano si la lectura ha sido correcta.

Para desarrollar el bloque de captura para controlar la posición de descarga, primero debemos analizar cómo el imán de la punta del robot transporta el precinto.

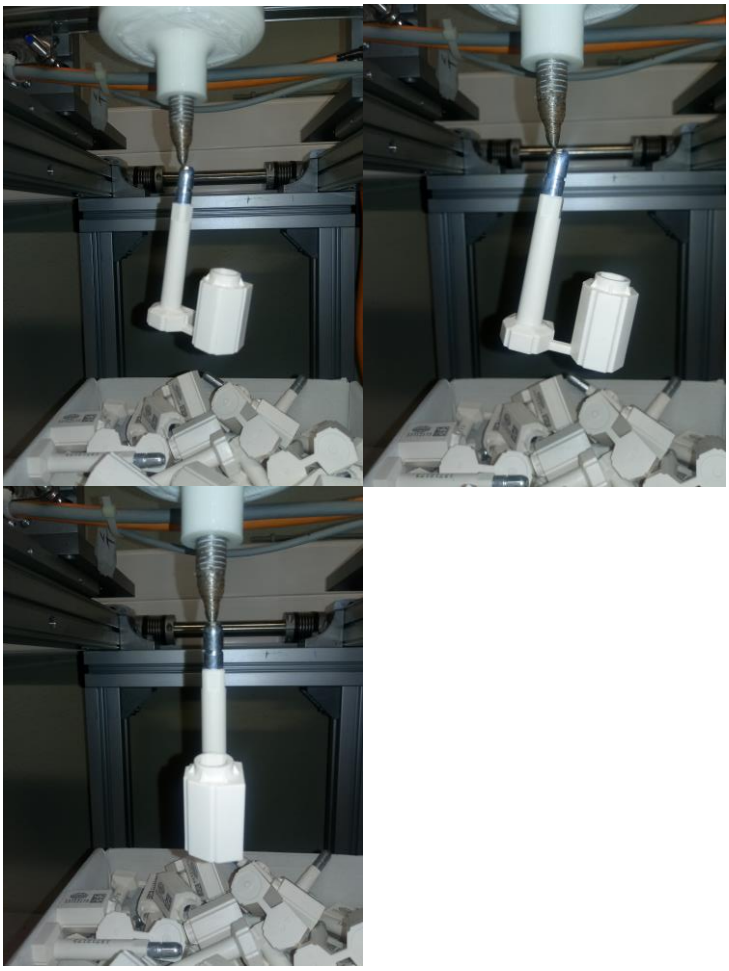


Imagen 8: detalle del transporte del precinto.

El precinto acaba colgado de la punta metálica, en una posición casi vertical, pero rotado de forma aleatoria. Esta rotación puede ser de entre 180º y -180º, con lo que tampoco se sabe en qué lado va a quedar el código. El precinto, en una posición casi vertical con la punta metálica hacia arriba nos viene bien, pues podemos descargarlo simplemente sobre una superficie plana y se equilibrará, pero la rotación es un problema. Va a ser necesario rotar el precinto antes de dejarlo en la superficie plana. Lo rotaremos hasta que el código quede plano contra la cámara que instalemos.

Para rotar el precinto vamos a necesitar usar algún sistema pasivo, pues queremos mantener coste y complejidad del equipo al mínimo por cuestiones de mantenimiento.

La solución a la que llegamos fue hacer pasar el precinto por un embudo, que tenga entrada circular y salida rectangular. Esta solución tiene un problema, y es que no podemos garantizar si el código numérico va a quedar a un lado u a otro, pero es algo que trataremos después.

El rectángulo de salida será el menor que acomode el precinto, y el diámetro de la circunferencia de entrada será la diagonal del rectángulo, por lo que cuando baje el precinto por el embudo, se verá forzado a alinearse con el rectángulo de salida.

Este primer diseño se realizó con una herramienta común en los programas de diseño 3D, que une dos planos, en el programa utilizado, Fusion 360, se llama solevar. Una vez diseñado se mandó fabricar por impresión 3D en PLA, y se detectaron una serie de problemas en las pruebas con la máquina.

- El primero de los problemas es que, al engancharse el precinto en la punta imantada, no lo hace siempre exactamente igual, con lo que se desplaza unos milímetros del centro, y en algunas descargas el precinto se quedaba enganchado en el borde del embudo.
- El segundo de los problemas es que, si el precinto descendía casi perpendicular al rectángulo de salida, se quedaba enganchado en las paredes del embudo, en vez de deslizarse por ellas para alinearse.
- El tercer problema es que, por inclinación y fricción contra la pared del embudo, el precinto podía quedar enganchado una vez alineado, como se ve en la imagen

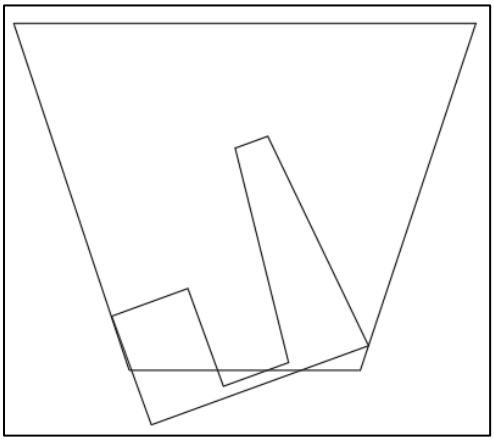


Imagen 9: croquis precinto enganchado.

El primer problema es fácil de solucionar: ampliamos el diámetro de la circunferencia. Nuestra limitación real está en el mínimo, que debe ser la diagonal del rectángulo, así que lo ampliamos varios centímetros para que haya margen suficiente.

El segundo es más difícil de solucionar, porque es necesario forzar la rotación en una situación en la cual el precinto es perpendicular a la pared del embudo, con lo que sería necesario forzar en la forma del embudo una especie de espiral. Al usar la herramienta de

solevado en el Fusion 360 se generan unas líneas que unen puntos clave de los dos planos que se unen, que son manipulables, y el programa las usa para definir la superficie a generar. Éstas, con nuestros dos planos, se generan entre las esquinas del rectángulo y cuatro puntos equidistantes entre en la circunferencia. Al rotar estos cuatro puntos se genera una ligera espiral en la superficie entre ambos planos. Usaremos esto para intentar evitar este problema, y de no solucionarse se intentarán otras alternativas.

El tercer problema se puede resolver alargando el rectángulo para que su lado largo sea igual a la longitud máxima que se forma al inclinarse el precinto.

Tras estos cambios en el diseño, se volvió a fabricar el embudo para realizar pruebas, y durante las mismas se vio que se habían conseguido solucionar los tres problemas planteados.

Imagen 10: Embudo bloque captura.

Ahora que tenemos un embudo funcional, hay que plantear que hacemos con el precinto tras pasarlo por el embudo.

En un principio debería bastar con poner una superficie plana a la suficiente distancia del final del embudo como para que el código numérico fuese visible, pero aparecieron una serie de inconvenientes al llegar a ese punto. La rotación del precinto al ser recogido es aleatoria, pues depende de cómo se pegue el precinto a la punta imantada, pero una vez se adhiere no cambia, y cuando corregimos esta rotación con el embudo no solucionamos el problema desde la raíz, así que en cuanto pasa del embudo vuelve a su posición original, así que necesitamos que el momento en el cual apagamos el electroimán sea uno en el que todavía estamos corrigiendo la rotación.

Al hacer esto vimos que, aunque la distancia a la cual dejamos caer el precinto es pequeña, unos 4 centímetros, al estar inclinado y no perfectamente vertical, en el rebote se gira si tiene espacio.

La solución a esto, que también nos permite dejar caer el precinto desde menos altura, es descargarlo en un espacio confinado con el sitio justo para que quepa a lo ancho. A este espacio le haremos un hueco por donde se pueda ver desde fuera el código, para poder captarlo con el sistema de visión que instalemos.

Hacer este hueco nos lleva de vuelta al problema de no poder garantizar en qué lado va a quedar el código numérico, y lo vamos a solucionar haciendo un hueco en cada lado, y poniendo una cámara a cada lado. Al decidir esto tuvimos en cuenta el material disponible, y adoptamos esta solución porque se disponía de cámaras suficientes, y las cámaras necesarias para este sistema son muy baratas.

Antes de plantear la tercera condición del bloque de captura, vamos a hacer una cosa con el espacio confinado que será útil para el sistema de visión cuando se diseñe: vamos a poner una rampa en uno de los lados.

Esta rampa tiene como objetivo corregir la posición, a lo largo del rectángulo del embudo, que ampliamos para evitar enganches, y provoca que en la descarga la posición, si bien no rotada, no es siempre la misma. Esta rampa forzará al precinto a acabar en una posición concreta, y eso además nos permite reducir el tamaño de los huecos que necesitamos para hacer visible el código numérico.

Ahora que tenemos completamente planteado el bloque de captura, aparece el problema que nos lleva a la tercera condición: el precinto se encuentra en un espacio confinado, con lo que no se puede acceder a recogerlo.

La tercera condición será que el bloque de captura necesita una forma de liberar el precinto una vez se extraiga el código numérico. Esta extracción no tiene condiciones, porque ya no es necesario que el precinto acabe en una posición concreta.

La forma más simple de conseguir esto sería instalar una trampilla justo donde se encuentra apoyado el precinto, para que caiga a un receptáculo secundario, porque modificar lo que hemos hecho hasta ahora para añadir una puerta por la cual se pueda

llegar a introducir una mano supondría añadir piezas móviles al espacio confinado y al embudo.

Para la trampilla se podría instalar un servomotor o un motor paso a paso, pero son componentes que requieren de bastante mantenimiento y es fácil que se estropeen, así que se optó por solenoides lineales.

En un solenoide lineal la bobina tiene forma cilíndrica y tiene un hueco en el centro para una barra de material ferroso, cuando pasa corriente la bobina genera un campo magnético y tira de la barra hacia el interior de la bobina. Utilizando dos de estos se puede hacer una trampilla, uno para la apertura y otro para el cierre, y quitando los cables no hay piezas que tiendan a estropearse a estos precios, como los engranajes de un servomotor, y el control es más simple que el de un motor paso a paso.

Para hacer esta trampilla quitamos el suelo donde se apoya el precinto al ser descargado, hacemos una superficie que vaya unida a las barras de los solenoides, y ponemos el soporte necesario para estos en el bloque de captura.

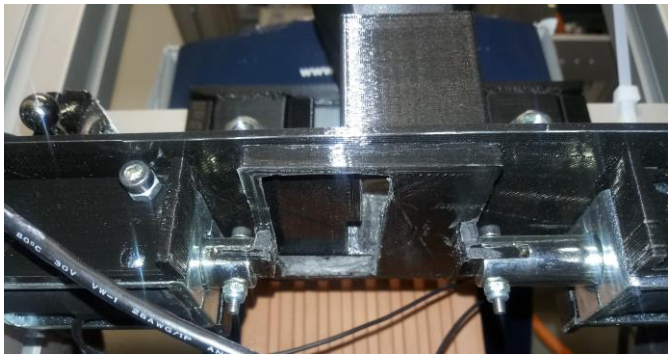


Imagen 11: trampilla

Elección del sistema de control

El sistema de control va a comprender tanto los programas para el control de los diferentes mecanismos como los aparatos que se utilicen. Las decisiones acerca de éste se van a tomar teniendo en cuenta las necesidades de las partes ya escogidas, como las guías lineales, y el conocimiento previo de programas por parte del alumno.

Para el control de las guías lineales va a ser necesario un ordenador, pues se controlan desde un driver que se comunica por puerto serie, y se descarta hacer todo el desarrollo en un microcontrolador.

Las cámaras de las que se dispone son webcams que se conectan con un cable USB, con lo que se refuerza la idea del uso del ordenador.

Para el control del motor paso a paso son necesarias salidas a 5V, con lo que va a ser necesaria una tarjeta de adquisición.

El control del electroimán de la punta del robot cartesiano y los solenoides lineales necesitan alimentaciones propias, con lo que la forma más fácil de controlarlos sería con un relé conectado a la tarjeta de adquisición que ya necesitamos.

En cuanto a los programas, será necesario programar los algoritmos tanto de visión como de control y comunicación de los motores. Se podría realizar toda la programación en un lenguaje como C++, pero el programa *Sherlock* de *Teledyne Dalsa* es parte del temario de una de las asignaturas de mención de la carrera, y dispone de algoritmos de visión y tratamiento de imagen, además de comunicación tanto por puerto serie como TCP-IP, y un entorno básico para programar, con lo que resulta perfecto para la aplicación.

Sherlock tiene licencia gratuita de prueba con algunas limitaciones, lo que permite mantener los costes bajos, pero una de esas limitaciones es que no se pueden adquirir imágenes desde el programa. Es posible tomarlas desde fuera del programa y cargarlas en el momento necesario, con lo que no será un problema.

En cuanto al ordenador, cualquier ordenador con sistema operativo Windows, para poder instalar *Sherlock*, y los puertos necesarios, sirve.

Una tarjeta de adquisición industrial es muy cara, pero un microcontrolador puede sustituirla. La decisión de cuál utilizar viene determinada por los conocimientos adquiridos, y por la gran cantidad de material que existe en internet sobre éste, siendo una placa de *Arduino* el microcontrolador que se escoge.

El sistema de visión se programará en *Sherlock* desde el ordenador principal, se comunicarán las órdenes para las guías lineales desde éste, y se coordinará con el *Arduino* para controlar el electroimán, los solenoides lineales y el motor paso a paso que controla el eje Z del robot cartesiano.

Diseño de los sistemas de visión

Vamos a necesitar dos sistemas de visión, uno para localizar precintos en el recipiente, y otro para leer el código numérico de los precintos.

Se van a utilizar tres webcams para los sistemas de visión, para reducir costes y por facilidad de uso.

Las tres cámaras poseen sistemas de iluminación propios. Dos de ellas van a tener enfoque fijo manual, y se utilizarán para la lectura de códigos en los precintos, ya que la distancia a la cámara de los precintos va a ser constante. La cámara que se va a utilizar para la caja de precintos sí tendrá enfoque automático, para tener en cuenta la distancia cambiante a los precintos, ya que conforme se vacía el recipiente la distancia a la cámara aumenta.

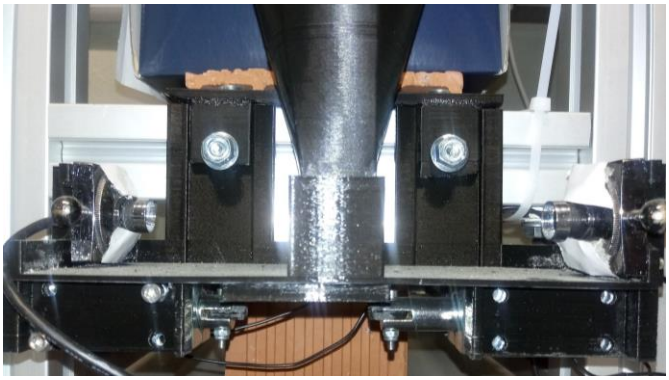


Imagen 12: Detalle de las cámaras del bloque captura.

A la hora de diseñar los sistemas de visión lo que vamos a estar decidiendo es el *FOV* (Field of View), y por ello donde colocar la cámara, la configuración de los parámetros de las cámaras y la iluminación. Los parámetros de las cámaras que vamos a modificar son brillo, contraste y tiempo de exposición.

A la hora de definir el *FOV* de nuestras cámaras tendremos en cuenta que no es completamente necesario ajustarlo por completo al objeto que se intenta ver, pues en el procesado de la imagen se pueden elegir regiones de interés dentro de la imagen, y por ello se podría escoger un *FOV* que recogiese mucha información innecesaria para conseguir ventajas en otros campos.

Empezando con el sistema de lectura de códigos, lo primero que se hizo fue, utilizando un programa para tomar video usando la webcam, ver la distancia en la que el *FOV* encuadra justo el código numérico, y tomamos una foto en esa posición para hacer pruebas. De momento utilizamos luz ambiente y dejamos los ajustes de la cámara en los predeterminados. Como de primeras pretendíamos utilizar luz ambiente, aún no habíamos intentado encender el sistema de iluminación de la webcam, pero notamos que, en ciertos momentos, cuando la cantidad de luz en lo que se enfocaba era reducida, las luces de la webcam se encendían solas, lo cual significa que son automáticas.

En *Sherlock* preparamos un programa básico para probar las imágenes que se extraen, para lo cual utilizamos la primera imagen que tomamos. Con esto probamos a tomar otras fotos, cambiando los ajustes de la cámara, y es cuando se notó que cuando el tiempo de exposición del sensor no está puesto en automático, y se fija un valor, la iluminación de la cámara se mantiene apagada, lo cual nos dice que ligados.

Siendo que la iluminación de la cámara no se puede encender sin poner el tiempo de exposición en automático, para las pruebas con los ajustes fijos del tiempo de exposición se iluminó de forma independiente.

Tras las diferentes pruebas se llegaron a diferentes combinaciones de parámetros e iluminaciones que daban buenos resultados, siendo una de ellas con el tiempo de exposición de la cámara en automático. Como en este caso no es necesario añadir un sistema de iluminación independiente, se tomó la decisión de utilizar esta configuración para este sistema de visión y añadir al bloque de captura un soporte a la distancia medida para las cámaras.

Para el otro sistema de visión comenzamos haciendo pruebas similares, pero el *FOV* de esta cámara es mayor que el de las utilizadas en la lectura de códigos, y se volvió preocupante la posibilidad de que se distorsionase la imagen al enfocar de cerca.

Para que una cámara tenga un *FOV* tan amplio necesita tener una lente ojo de pez, y éstas tienen la tendencia a distorsionar la imagen a cortas distancias, con lo que se podía convertir en un problema. Para evitar esta distorsión es necesario alejar la cámara de lo que se intenta visualizar, pero como después de tomar la imagen podemos escoger la región de interés en el programa que se haga en *Sherlock*, se descartó encajar el *FOV* con el recipiente de los precintos.

Con la posible distorsión de la imagen en mente se planteó en qué posibles lugares del robot se podía colocar la cámara. El lugar más adecuado era junto al motor paso a paso del eje Z, y es el primero que se probó. El otro posible lugar donde colocar la cámara es sobre el brazo del eje Z, la pieza móvil que sube y baja, en un punto de este dónde el encuadre se considere correcto, que, aunque a menor distancia que junto al motor, mientras la distorsión no supusiese un problema era aceptable, pero colocar cámaras en piezas móviles no es recomendable, con lo que esta opción no se probó en un principio.

Para las pruebas con la cámara junto al motor se solicitó al servicio de diseño e impresión 3D un soporte que permitiese que la cámara estuviese alineada con el recipiente de los precintos.

Una vez puesta la cámara, cómo detectar el punto de interés, la punta metálica, fue el siguiente tema a tratar.

Lo primero que se intentó fue detectar la diferencia de colores con iluminación ambiente. En *Sherlock* hay un algoritmo que se puede entrenar para clasificar zonas de la imagen por colores, usando los valores de los píxeles. Se trató de asociar, utilizando una imagen de prueba, los valores de los pixeles en las puntas con el "gris", y el del resto del precinto como "marrón".

Al probar esto no fue posible asociar los valores de manera que se formasen regiones consistentes en las puntas, y que no se diesen casos de falsos positivos en otras zonas, por las zonas brillantes causadas por el reflejo de la luz.

Para intentar corregir esto se intentó iluminar de forma artificial, apagando las luces del laboratorio y usando una iluminación difuminada para evitar brillos directos, pero con el material disponible no fue posible conseguir un resultado satisfactorio, y se tuvo que descartar la idea de detectar las puntas usando color.

La idea que surgió fue utilizar justo lo que había imposibilitado detectar por colores, los reflejos. Esto de primeras tiene un problema, y es que también hay reflejos fuera de las puntas metálicas, pero las puntas reflejan mucho más que el cuerpo del precinto, aunque a simple vista la diferencia no es notoria. Al ajustar el valor del tiempo de exposición de una cámara puedes controlar cuanta luz recoge el sensor de ésta. Con tiempos muy bajos solo brillos muy fuertes o iluminación directa al sensor consiguen hacer reaccionar el sensor. Lo que se planteó fue iluminar con unas lámparas LED muy potentes, para causar la máxima cantidad de brillos, y reducir el tiempo de exposición lo suficiente como para que solo los brillos de las puntas fuesen visibles. Se ajustó también el valor del contraste para maximizar la diferencia entre las zonas con mucho reflejo y las que no.

Con estos cambios se hicieron pruebas, y se consiguieron detectar puntas, pero los resultados no fueron los esperados al comparar con lo que se veía a simple vista. Al observar el recipiente de los precintos se veían todas las puntas visibles brillando, pero solo algunas de estas aparecían en la imagen de la cámara. Buscando los motivos se

encontró un patrón al hacer pruebas grabando video con la cámara en esta configuración: las puntas visibles con la cámara no lo eran en ciertos ángulos. Esta teoría se pudo comprobar haciendo pruebas grabando video y moviendo un precinto a mano.

La explicación que se encontró a este fenómeno fue la reflexión de la luz. La cámara al solo tener un punto de entrada de luz los rayos de luz que se pueden recoger son limitados, y con el tiempo de exposición tan bajo para evitar falsos positivos en el cuerpo del precinto significa que solo los reflejos más directos son detectados por el sensor.

La iluminación instalada en este momento eran dos lámparas LED colocadas una a cada lado del recipiente, debajo de las guías del eje X, y para que bañasen el recipiente completo con luz estaban inclinadas, lo cual estaba afectando al ángulo de los rayos reflejados.

Lo primero que se intentó fue instalar otra iluminación. Se optó por un diodo LED pequeño muy potente que no se habían gastado por ser más difícil de colocar sobre la estructura que las lámparas, que solo requerían de tornillería. Se probó a colocar uno de este LED justo en vertical, alineado con la cámara, en el cabezal. Los resultados mejoraron, pero los precintos exteriores seguían siendo visibles en pocas situaciones. Con iluminación y cámara fijos no se iba a poder solucionar el problema.

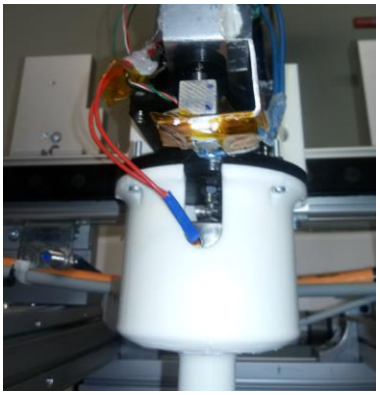


Imagen 13: Colocación de la iluminación LED.

Aquí es donde se consideró el otro posible punto en el que instalar la cámara. Con la cámara tan lejos como la teníamos, mover el brazo Z en X e Y supone un cambio en el ángulo de incidencia demasiado pequeño, pero colocándola en el punto más bajo que no dé problemas de distorsión y encuadre, el recipiente completo haría que el cambio en el ángulo de incidencia fuese mayor, y con ello al cambiar de posición el brazo Z haría que los rayos de luz recogidos tuviesen mayor variedad de ángulos, al hacer los movimientos.

Se hizo un soporte con una chapa de aluminio a la que se le hicieron los agujeros necesarios, y se hicieron pruebas.

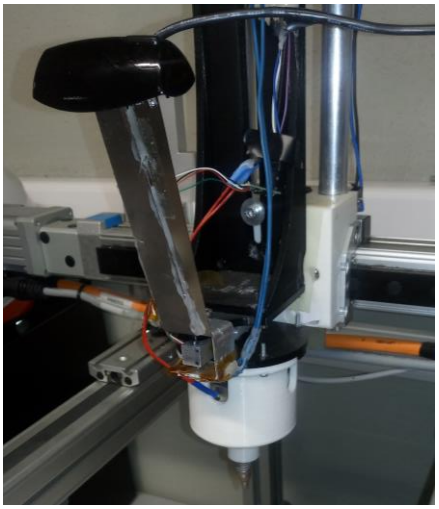


Imagen 14: posición de la cámara de la caja.

Como se esperaba, en fotos individuales los resultados tenían los mismos problemas, pero lo que igual no se veía en la foto de la posición 1 se veía en la foto de la posición 2.

La solución final fue crear una matriz de posiciones, y cuando el programa detectaba que en la primera posición no se encontraban puntas de precinto, avanzaba a la siguiente, hasta encontrar una posición de la matriz en la cual si se veían reflejos.

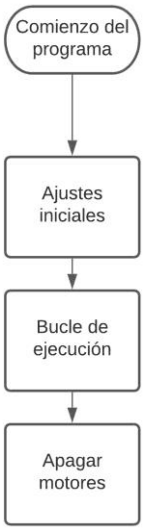
Programación del sistema de control

Ahora que se han explicado las líneas de pensamiento que han llevado al sistema final, falta una explicación en detalle del sistema de control del prototipo y los programas desarrollados para su funcionamiento.

Una versión final de la dispensadora de precintos planteada en este proyecto requeriría de un solo programa por el uso de una tarjeta de adquisición, pero este prototipo depende de 3 programas: uno principal en el PC en *Sherlock* que controla el flujo del programa, uno secundario en ~~el~~ *Arduino* que hace de tarjeta de entradas y salidas, y unos pocos cálculos, y un tercer programa en el PC en *Python* que servirá para parchear la limitación que supone la licencia de prueba de *Sherlock*.

El parche se basa en lo siguiente: en *Sherlock* hay ventanas de imagen, en las cuales se carga la imagen para trabajar sobre ella. En estas ventanas se puede escoger qué imagen cargan de entre unas cuantas posibilidades, la relevante para el parche es cargar un archivo desde el almacenamiento del PC, para lo cual se introduce el directorio y el nombre. Cada vez que se ejecuta la ventana de imagen en el programa de *Sherlock* con esta opción puesta vuelve a cargar el archivo que le hemos indicado, aunque no sea el mismo. El parche usa un programa aparte, que toma las imágenes, las guarda con el nombre y directorio indicados en la ventana de imagen, y después en el programa de *Sherlock* se ejecuta la ventana de imagen, que ahora coge la nueva imagen. Todo esto se coordina utilizando comunicación TCP-IP entre ambos programas.

Programa *Sherlock*



Flujograma 1: Programa completo

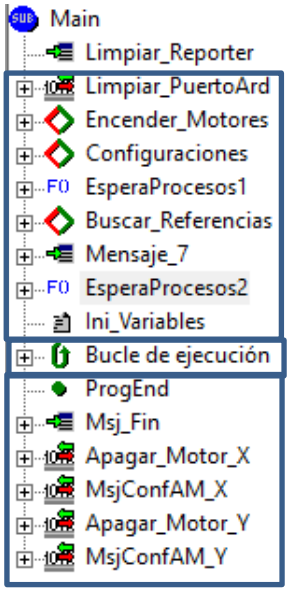
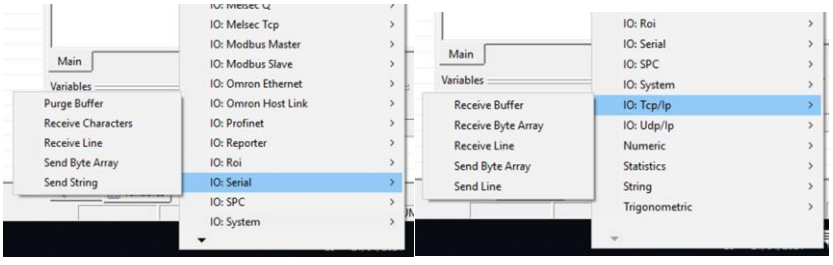


Imagen 15: Programa completo cerrado

El programa se puede separar en 3 partes principales: unos ajustes iniciales que ejecutan una vez al iniciar el programa, el bucle de ejecución donde se espera la solicitud de precinto y se produce la extracción, y una pequeña fase final de apagado de motores que se ejecuta cuando se sale del bucle.

El programa contiene muchas funciones de comunicación para enviar las ordenes a los motores, las comunicaciones con el *Arduino* y las ordenes de tomar fotos del programa parche. Las funciones utilizadas son casi todo el tiempo las mismas, asi que en vez de explicar cada una que aparezca, explicar como funcionan y luego comentar lo que cambia será más eficiente.

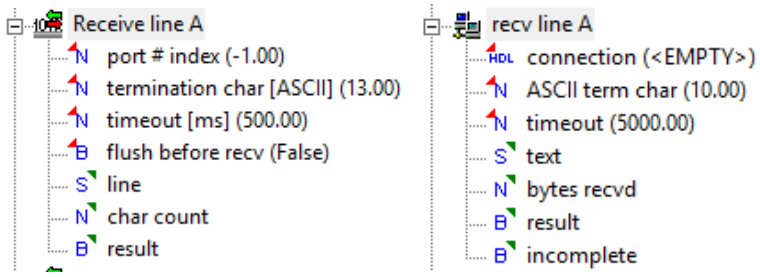


a b

Imagen 16: Funciones comunicación serie (a), funciones comunicación TCP-IP (b)

De las opciones que ofrece *Sherlock* para las comunicaciones utilizamos “Recibir línea” tanto para TCP-IP como para Serie, que recogen lo que haya en el buffer de comunicaciones hasta encontrar un símbolo concreto, y para enviar información se usará

"Enviar string" para Serie, y "Enviar línea" para TCP-IP, que funcionan de forma practicamente idéntica, pues ambos tienen como mensaje ~~a~~ enviar un *string.*



a b

Imagen 17: Funciones recibir línea. Comunicación serie (a), TCP-IP (b)

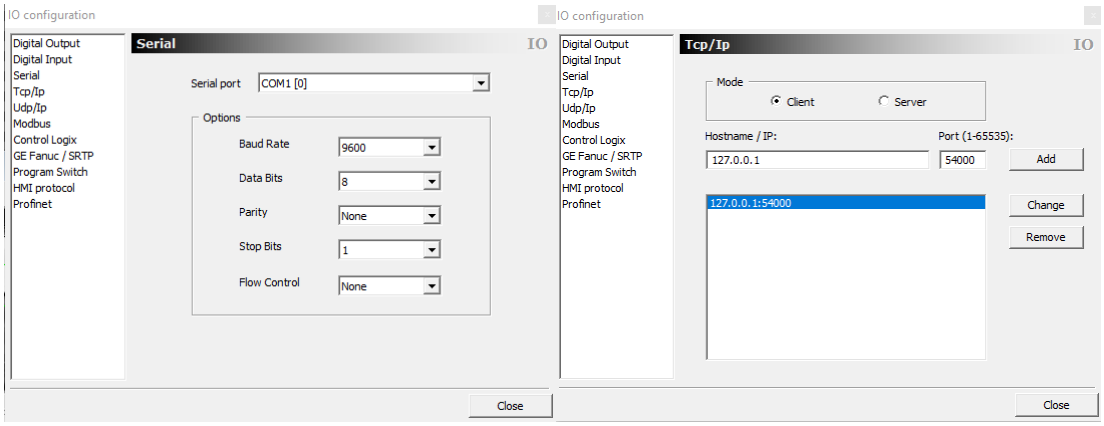
A la hora de utilizar las funciones de recibir línea hay que introducir 3 datos:

- Puerto/conexión por la cual queremos enviar el mensaje
- El valor ASCII del símbolo que estamos esperando al final del mensaje.
- Y un valor de tiempo de espera, que define cuanto tiempo puede pasar desde que se ejecuta la función hasta que se considera que no ha habido comunicación.

El valor ASCII que vamos a buscar es el 13 siempre que se usen las funciones de recepción de línea, que es "carriage return", o salto de línea. Los controladores de los motores envian siempre un salto de línea despues de cada mensaje, y en *Arduino* se puede programar que también lo haga, con lo que al buscar ese símbolo siempre recogeremos los mensajes completos.

Un detalle sobre las funciones de recibir línea, el símbolo indicado no se incluye en el mensaje que se guarda.

En las funciones de envio solo es necesario introducir puerto/conexión y mensaje.



a b

Imagen 18: Configuración comunicación serie (a). Configuración TCP-IP (b)

En el menú de configuraciones de entradas y salidas de *Sherlock* se pueden configurar las comunicaciones serie y TCP-IP. En el caso de las comunicaciones serie, dejamos los valores predeterminados, y revisamos las conexiones que se disponen. En este PC el motor X ha aparecido como COM1(0), el motor Y como COM3(2) y ~~el~~ *Arduino* como COM4(3). No hay forma de cambiarlo desde *Sherlock*, así que se trabaja alrededor de como han sido denominados por el sistema operativo. Para ver qué puerto era cada motor fue necesario enviar mensajes de prueba, y ver cual de los motores reaccionaba, al no haber datos revisables.

En el caso de la comunicación TCP-IP, definiremos *Sherlock* como el cliente, y nos conectaremos a la dirección 127.0.0.1, que es la dirección para comunicarse con el mismo ordenador. Lo hacemos por el puerto 54000, para asegurar evitar coger un puerto ocupado por otra tarea del ordenador. Dentro del programa en *Python* que toma las imágenes el "1" se refiere a la cámara de la caja de los precintos, el "2" a la cámara de la izquierda del bloque de captura y el "3" a la cámara derecha.

En la imagen 15 se ha recuadrado qué funciones del programa componen cada uno de los bloques del diagrama. Hay una funcion que se ha dejado fuera, "Limpiar_*reporter*", por ser algo independiente al funcionamiento del programa.

El *reporter* de *Sherlock* es una ventana por la cual se pueden imprimir mensajes, y no se vacia entre ejecuciones. La función "Limpiar_*reporter*" se ejecuta una vez, al principio, para vaciarla.

Ajustes iniciales

En ajustes iniciales se van a hacer las configuraciones necesarias de los motores de los ejes X e Y, pues su configuración no se guarda cuando son apagados, se va a preparar la comunicación con el *Arduino*, y se van a llevar los 3 ejes a referencia.

Flujograma 2: Ajustes iniciales

Lo primero que se va a hacer es limpiar el buffer de la comunicación serie con el-*Arduino*. Esto es necesario porque *Arduino*, al encenderse, envía una serie de mensajes para confirmar conexión, y si se quedan en el buffer cuando se revise para alguna tarea del programa dará problemas, pues hay saltos de línea en el mensaje que envía.

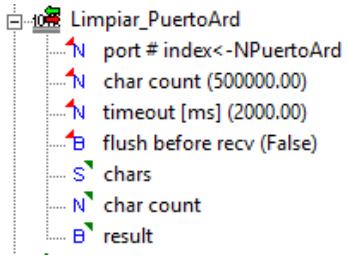


Imagen 19: Función limpiar buffer comunicación serie con *Arduino*

La función de *Sherlock* que se utiliza aquí es "recibir caracteres", pues no tenemos un símbolo concreto a buscar que solo ocurre al final del mensaje, y no utilizamos la función de limpiar el buffer porque en caso de encender *Arduino* y el programa al mismo tiempo, *Arduino* podría estar todavía enviando parte de este mensaje cuando se limpia el buffer, llenándolo de nuevo.

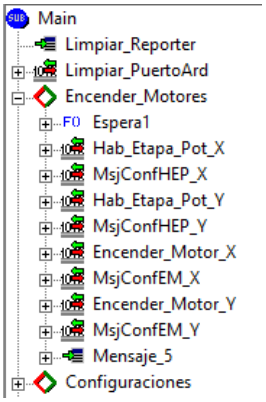
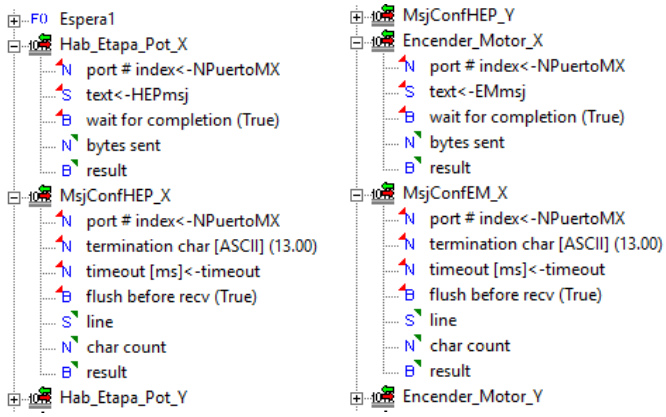


Imagen 20: Tarea encendido de motores desplegada

Un detalle que se repite a lo largo del programa: el desplegable de muchas funciones está hecho con una función "IF" de *Sherlock*. Esto se ha hecho así para poder organizar el código, ya que se volvería inmanejable pues no hay forma de colapsarlo, y la condición del "IF" es que una variable definida de antemano tenga un valor concreto, en el caso de la versión final 0, pero durante el desarrollo se modificó este valor para poder ejecutar solo algunas funciones.

En la tarea "Encender_Motores" se pone una pequeña espera para que el ordenador pueda hacer las comunicaciones internas necesarias con los drivers de los motores, y después se envían a ambos drivers dos mensajes: uno para habilitar la etapa de potencia, y otro para el propio encendido del motor.



a b

Imagen 21: Comunicaciones habilitar etapa de potencia (a). Comunicaciones encendido de motor (b)

Los mensajes para ambos motores son idénticos, por lo que en la imagen 20 solo aparecen las funciones de comunicación del motor X. Los mensajes están en hexadecimal y son específicos de los motores, así que, a simple vista son inentendibles. Todas las variables fijas, como estos mensajes, están en el anexo 1.

Las funciones de lectura de líneas se utilizan para limpiar el buffer, y para revisión manual de que todo funciona como es debido en caso de notar un comportamiento extraño en el robot.

Después de encender los motores configuramos una serie de valores internos de los drivers.

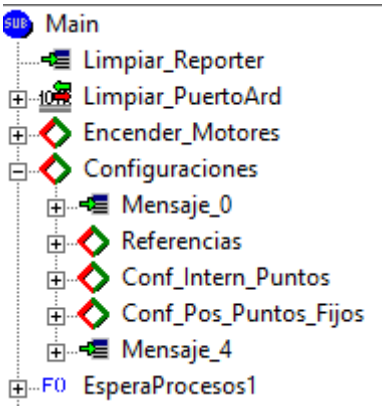
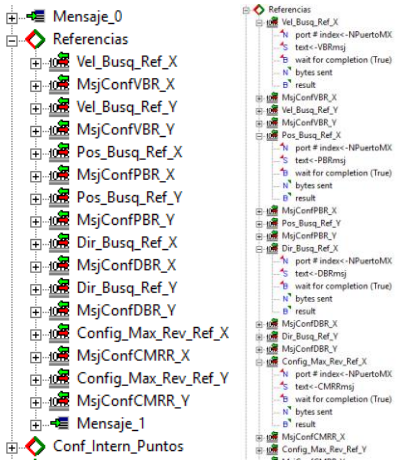


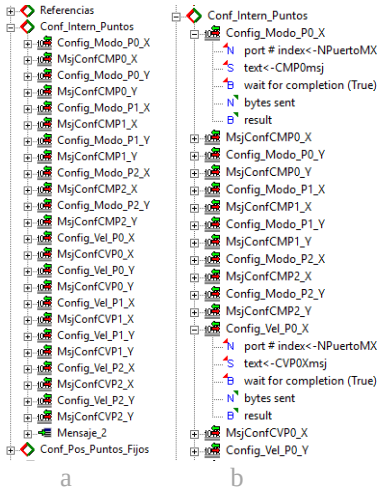
Imagen 22: Tarea Configuraciones desplegada



a b

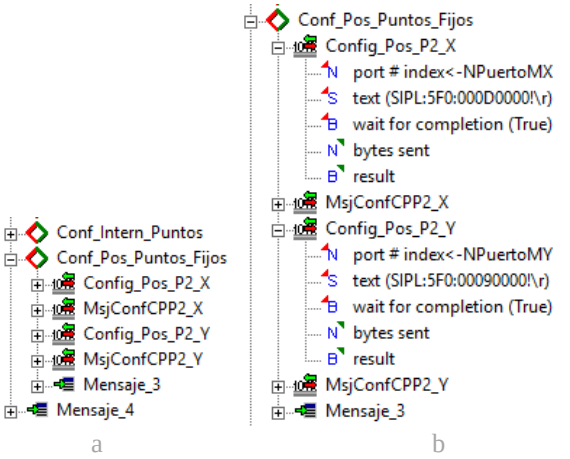
Imagen 23: Tarea referencias desplegada (a). Comunicaciones configuración referencias (b).

La búsqueda de las referencias de los motores tiene una serie de configuraciones necesarias antes de poder realizar la búsqueda. Estas son la velocidad a la cual van a moverse en busca de la referencia, la posición, que en nuestro caso será la búsqueda de 0, el sentido de la búsqueda, y un valor de revoluciones máximas que el motor puede realizar en busca de la referencia.



a b

Imagen 25: Tarea configuración interna de puntos desplegada (a). Comunicaciones configuración modo y velocidad de puntos (b)



a b

Imagen 24: Tarea configuración puntos fijos desplegada (a). Comunicaciones configuración posición P2 X e Y (b)

Los motores se controlan escogiendo un punto de una lista, y dándole la orden de ir a él. Estos puntos se han de configurar de forma previa.

Se utilizan 3 puntos en este programa:

- P0 es el punto donde se ha detectado precinto,
- P1 el punto donde se va a situar el robot para tomar foto,
- P2 es el punto de descarga.

Primero configuramos modo y velocidad de cada punto que se vaya a gastar, y después en los puntos que tengan una única posible posición a lo largo del programa configuramos la posición de destino.

En la imagen 24b se pueden ver los mensajes para configurar la posición de destino del punto 2. En estos se pueden ver 8 cifras no cortadas por un ":", ese es el valor en hexadecimal de la posición de destino. Esta posición está cm para el eje Y, y en revoluciones para el eje X. Cuando se hizo esta parte del código no se sabía todavía, lo que se había extraído es que 00090000 es la posición media del eje Y y 000D0000 es el valor máximo que el eje X es capaz de recorrer.

La última tarea de los ajustes iniciales es la búsqueda de referencias.

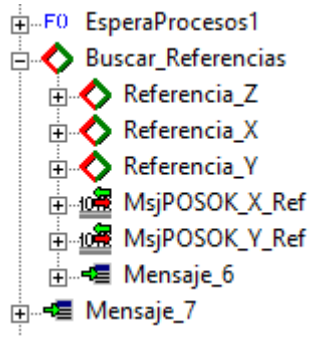


Imagen 26: Tarea búsqueda de referencias desplegada

Se realiza una espera para asegurar que los drivers no están ocupados tras las configuraciones, y se hacen las búsquedas en el orden que se ve en la imagen 25. Este orden es muy importante, pues en caso de haber sido necesario hacer una parada de emergencia, es posible que el eje Z se encontrase bajado. En caso de mover el eje X o Y con el eje Z bajado es muy probable causar daños al robot.

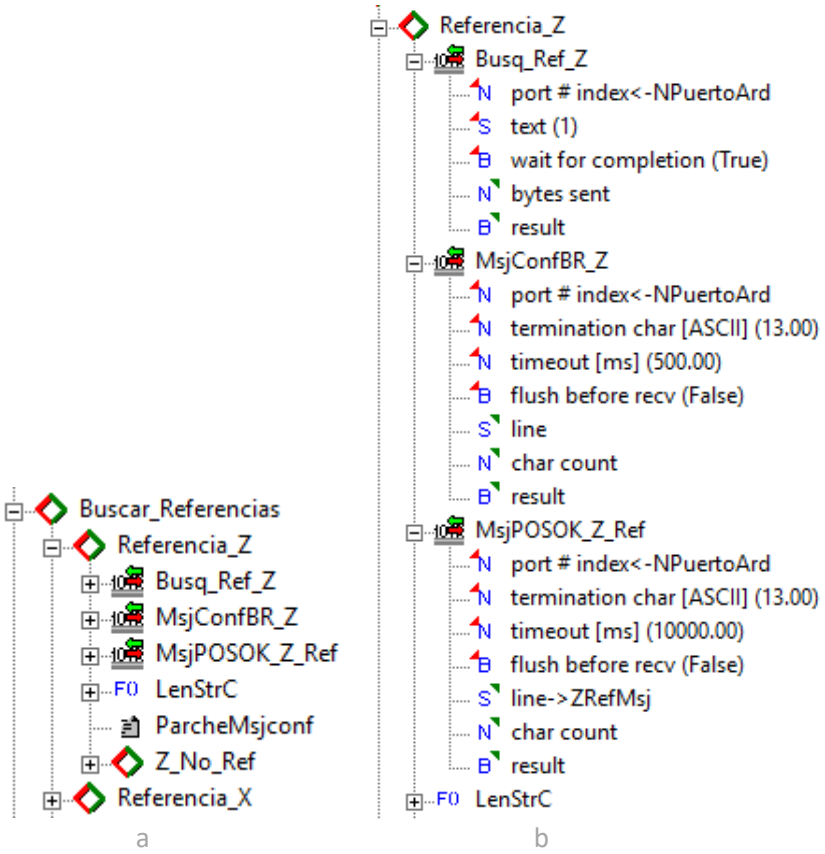


Imagen 27: Tarea búsqueda de referencias desplegada (a). Comunicaciones búsqueda de referencia Z (b)

En la búsqueda de referencias habrá dos mensajes de confirmación: el primero marca que se ha recibido la orden, y el segundo marca que se ha alcanzado la referencia.

El mensaje que se envía al *Arduino* para ir a referencia es un simple “1”. Las diferentes órdenes para el *Arduino* están codificadas con un simple número, ya que la cantidad de ordenes posibles es reducida y no es necesaria una mayor codificación.

No se han realizado comprobaciones para todos los posibles errores, pero algunos de ellos, si tenían el potencial de causar daños o fuese a ser necesario reaccionar muy rápido para hacer una parada de emergencia si tienen una comprobación y actuación en caso de error.

De normal el segundo mensaje de confirmación, el de posición de referencia alcanzada, se utiliza en el programa como una espera para dar tiempo al eje en cuestión a alcanzar la posición deseada, pero en casos como la vuelta a referencia del eje Z se analiza el mensaje para comprobar que todo ha funcionado.

El *string* enviado se guarda en la variable "ZRefMsj" para poder actuar sobre ella en un *script*, ya que *Sherlock* solo permite trabajar en un *script* con variables guardadas, y para que sea más fácil de localizar cuando *Sherlock* permite utilizar datos de las funciones, porque se acumulan muy rápido.

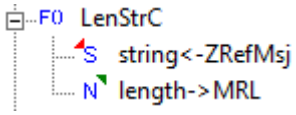


Imagen 28: Función lectura longitud de *string*

Antes de comprobar el mensaje en un *script*, extraemos su longitud y guardamos el valor en la variable "MRL". Extraemos la longitud en esta función de *Sherlock* en vez de directamente guardar el valor que la función "Recibir línea" proporciona porque extraemos este valor para un parche.

Este parche, que se hace en el *script* "ParcheMsjconf", se hace por un *bug* que apareció a mitad de desarrollo, del cual no se consiguió encontrar la causa por ser inconsistente. El *bug* consiste en que en las funciones "Recibir línea" de la comunicación con el *Arduino* a veces se guarda en el *script* un salto de línea al principio, a pesar de que la función no debería haber ninguno o parar al detectarlo. De normal se podría haber solucionado modificando la comprobación, que se hace con una comparación, añadiendo el salto de línea al *string* de comparación, pero *Sherlock* no lo reconoce de la misma manera.

La solución que se adoptó fue sustituir el valor del mensaje recibido con el valor del *string* en la posición final. El mensaje de comprobación solo tiene un carácter, con lo que recoger solo el último basta para parchear el *bug*.

```
Vars.ZRefMsj=Vars.ZRefMsj.charAt(Vars.MRL-1)
```

Imagen 29: Código parche mensaje *Arduino*

Los *script*s de *Sherlock* se programan en *JavaScript*, y los vectores en *JavaScript* comienzan por la posición 0, por ello se usa "MRL"-1 para coger el último carácter.

Una vez corregido el mensaje se ejecuta la función IF "Z_No_Ref", que compara el mensaje con la variable "CompZRef", que contiene el *string* que se espera recibir.

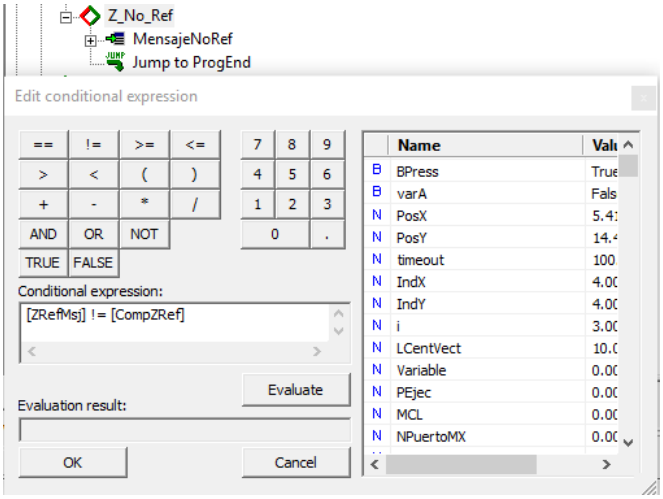


Imagen 30: Comparación función "Z_No_Ref"

El contenido se ejecuta si el mensaje recibido y la variable de comprobación no coinciden, cosa que pasaría de no haberse hecho el parche, o no llegase el mensaje por un problema de comunicación o que no hubiese llegado a referencia en el tiempo permitido.

El contenido es un mensaje por el *reporter* de *Sherlock* y una función "Salto" que lleva al programa al punto designado, en este caso "ProgEnd".

"ProgEnd" se encuentra justo después del bucle de ejecución, con lo que cuando se salta a este punto lo único que se ejecutan son las funciones del apartado "Apagar motores" del diagrama 1.

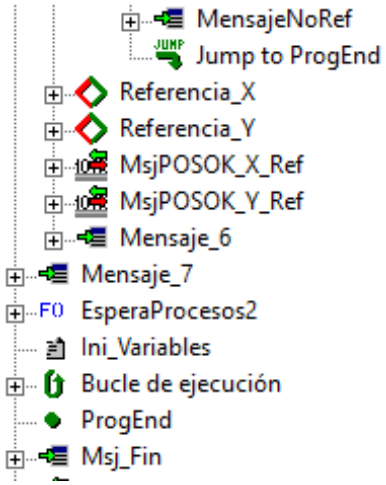


Imagen 31: Posición en código del salto a "ProgEnd"

Después de que el eje Z llegue a referencia se mandan a referencia los otros dos ejes, ahora que no hay peligro de daños en el robot.

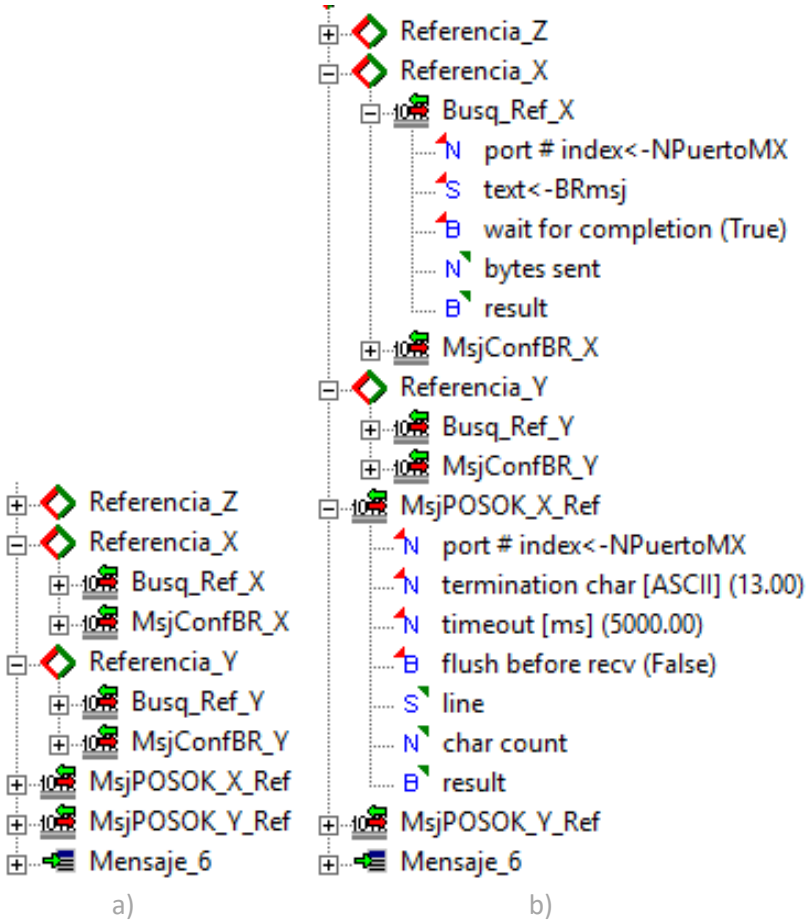


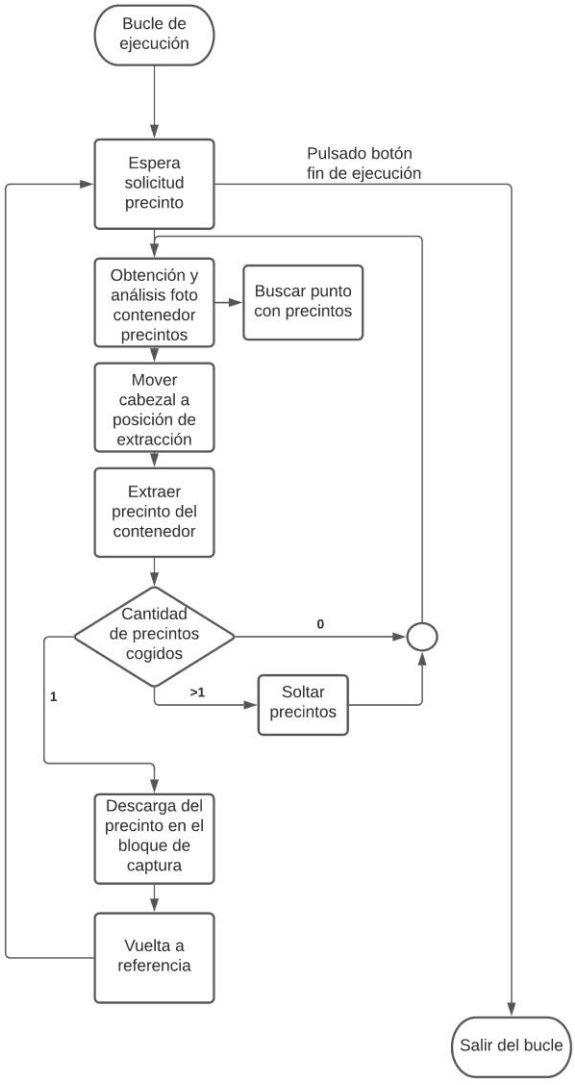
Imagen 32: Tarea búsqueda de referencias X e Y desplegadas (a). Comunicaciones búsqueda de referencia X e Y (b)

Para estos ejes, como no es necesario esperar a que uno termine para mover el otro sin peligro, se manda la orden de búsqueda de referencia a ambos y luego se ponen las esperas del mensaje de confirmación de que han llegado a la posición.

Antes de entrar en el bucle de ejecución se inicializan las variables que se van a utilizar en este a valores inocuos, pues *Sherlock* guarda los valores de la última ejecución realizada.

Bucle de ejecución

El bucle de ejecución es la parte del programa donde el robot se queda a la espera de una solicitud de precinto para comenzar el dispensado, que en este prototipo se ha simulado con un simple botón.



Flujograma 3: Bucle de ejecución

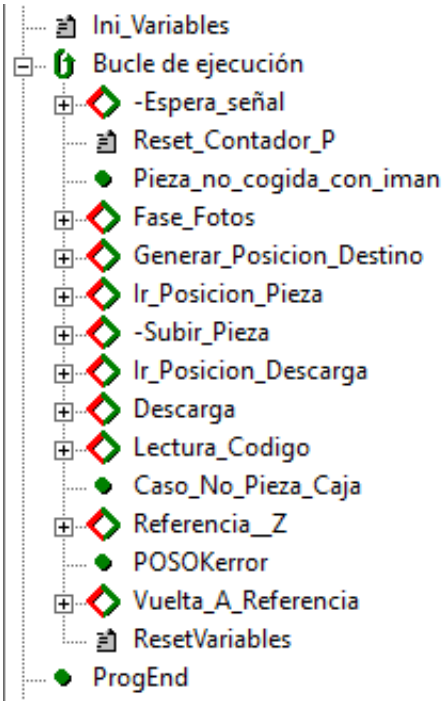


Imagen 33: Bucle de ejecución desplegado

El bucle de ejecución funciona como un "WHILE (True)", con lo que se ejecuta de forma indefinida, hasta que se hace un "Jump" a una parte del programa que se encuentre fuera del bucle.

El primer paso del bucle de ejecución es entrar en modo espera de solicitud.

Para entrar en este modo primero se envía una orden al *Arduino*, el mensaje "5", que pone al *Arduino* a la espera de la señal del botón para enviar a *Sherlock* el mensaje de que ha habido una solicitud.

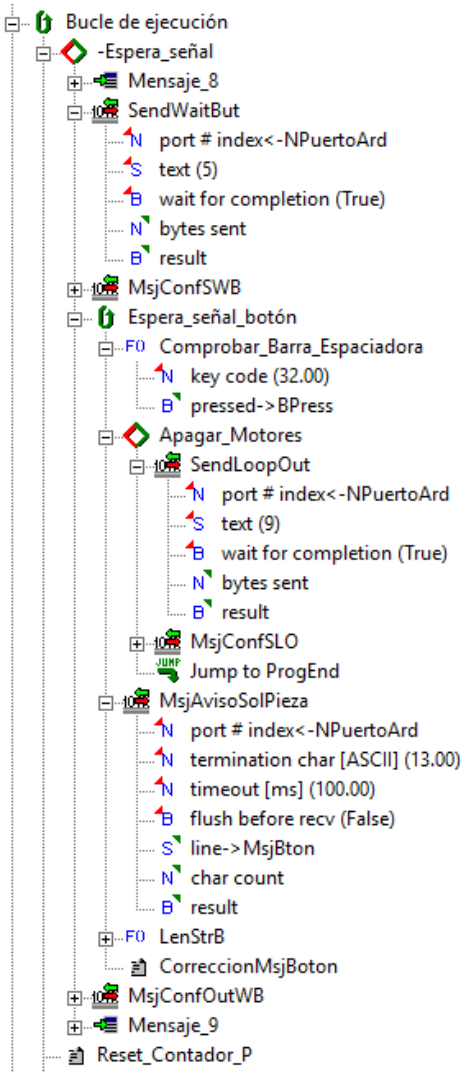


Imagen 34: Tarea "Espera_señal" desplegada

Después de poner al *Arduino* es modo espera el programa de *Sherlock* entra en el bucle "Espera_señal_botón", donde hace dos cosas. La primera es comprobar si se pulsa la barra espaciadora, que es el botón del teclado designado para salir del bucle de ejecución, y comprobar si llega desde la comunicación serie con el *Arduino* el mensaje de que ha habido una solicitud.

El bucle "Espera_señal_botón" tiene como condición de ejecución que el mensaje recogido del puerto serie del *Arduino* sea distinto a la variable "CompBton", que contiene el mensaje esperado, lo cual supone que el bucle se repite hasta que haya una solicitud. Este mensaje se revisa y corrige de la misma manera que el mensaje de llegada a referencia del eje Z, que se ha explicado anteriormente.

"Apagar_Motores" es un "IF" que comprueba si "BPress" tiene como valor "True". "BPress" es una variable donde se guarda el resultado de la función "Comprobar_Barra_Espaciadora", que recoge si se está pulsando o no la barra espaciadora cuando se ejecuta. Devuelve "True" cuando está siendo pulsada. En caso de que se cumpla esta condición, se envía al *Arduino* el mensaje "9" que le indica que salga del modo esperar solicitud, y se hace un "Jump" a "ProgEnd", que se encuentra justo después del bucle de ejecución.

Justo después de salir de "Espera_señal" se reinician a 0 los valores de las variables "NPuntosRecorridos" y "NIntentos" en el *script* "Reset_Contador_P". Estas dos variables sirven de contador de cuantas posiciones ha recorrido el robot en busca de una foto en la cual fuesen visibles precintos, y de contador de cuantas veces ha bajado a recoger un precinto y al subir no llevaba 1 precinto, respectivamente.

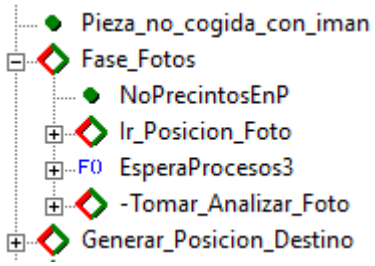
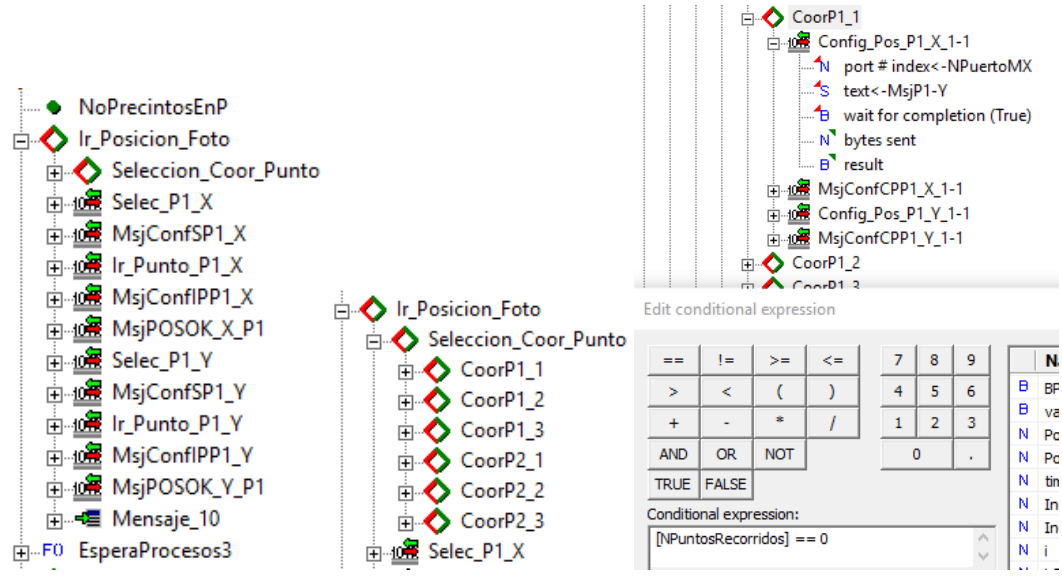


Imagen 35: Tarea "Fase_Fotos" desplegada

Una vez ha habido una solicitud de precinto es momento de localizar un precinto para poder recogerlo y dispensarlo. El programa ejecuta "Fase_Fotos", donde el robot coloca la cámara en posición de tomar foto, se toma una foto y se analiza para extraer la posición del precinto localizado.



a b c

Imagen 36: Tarea "Ir_Posición_Foto" desplegada (a). Selección de coordenadas para el punto P1 desplegada (b). Condición ejemplo de selección de coordenadas (c)

La posición de la cámara para tomar la foto no es siempre la misma, como se explicó en el diseño de los sistemas de visión. Dadas las limitaciones que imponen el *FOV*, el recorrido de los motores y el tamaño del recipiente, y ser este proyecto un prototipo para

poner a prueba una idea, se han generado 6 posiciones para la matriz que el sistema recorrerá en busca de un reflejo.

Cuál de las posiciones se escoge se decide con un contador, la variable "NPuntosRecorridos", y dentro de "Selección_Coor_Punto" se realizan una serie de "IF" que ejecutan una configuración de la posición del punto "P1" distinta dependiendo del valor de "NPuntosRecorridos".

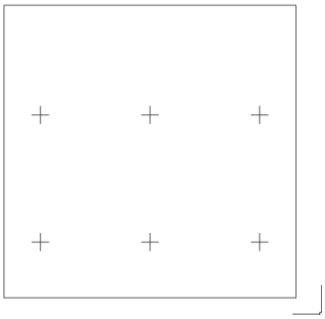


Imagen 37: Posiciones aproximadas sobre la caja de precintos

Una vez escogido a qué punto se va a ir de la lista se mandan las comunicaciones necesarias para el movimiento.

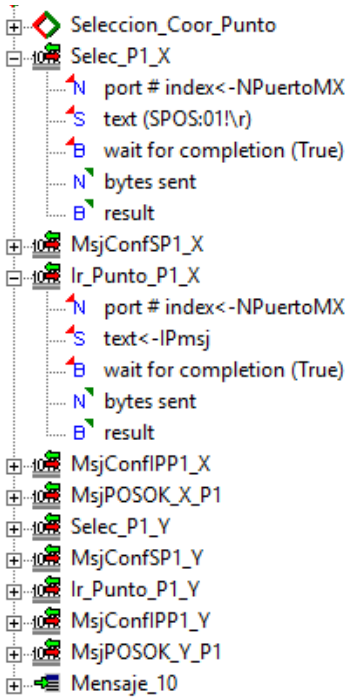
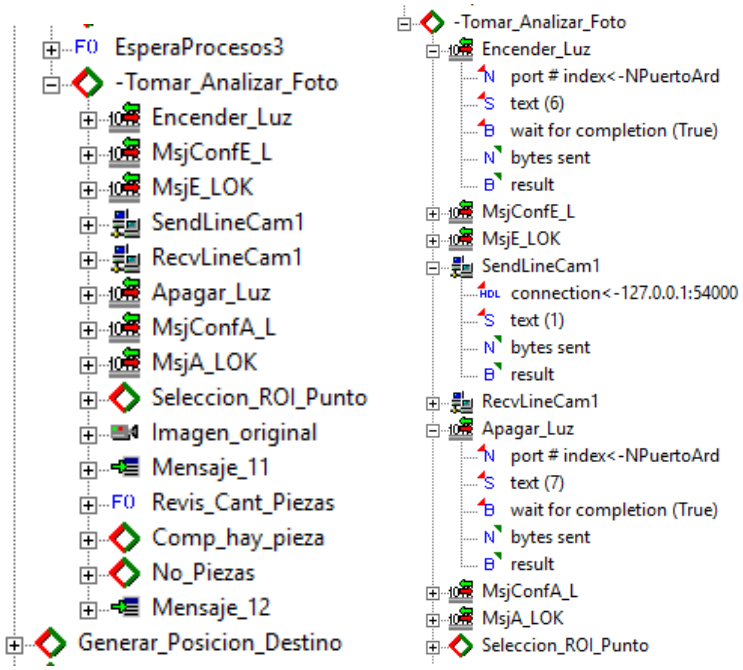


Imagen 38: Comunicaciones movimiento a punto P1

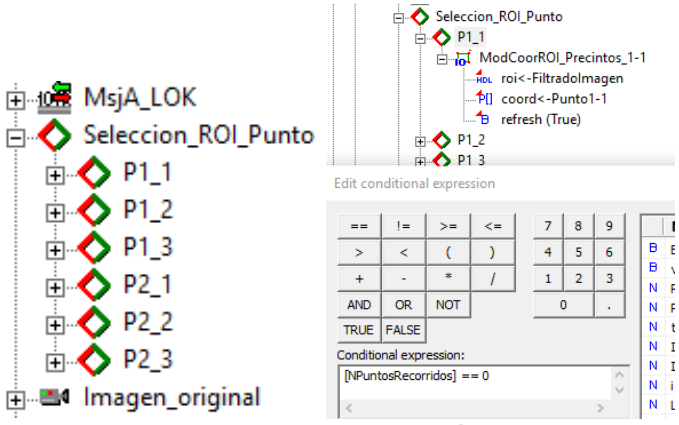
Una vez el robot coloca la cámara en posición se entra en “Tomar_Analizar_Foto”.



a b

Imagen 39: Tarea “Tomar_Analizar_Foto” desplegada (a). Comunicaciones toma de foto de la caja de precintos (b)

Aquí se toma y después procesa la imagen. Lo primero que se hace es encender la luz enviando la orden “6” al *Arduino*, con dos mensajes de comprobación que sirven tanto para evitar posibles problemas de comunicación como para dar un pequeño espacio de tiempo al LED para que se caliente, después se envía por TCP-IP un “1”, que indica al programa auxiliar que tome foto con la cámara 1, y finalmente se apaga la luz enviando al *Arduino* un “7”.



a b

Imagen 40: Selección de coordenadas para el ROI desplegada (a). Condición ejemplo de selección de coordenadas (b)

La imagen está tomada, ahora de la misma manera que con la posición para tomar la foto escogemos unas coordenadas para la región de interés (ROI) de la imagen sobre la que trabajar, ya que no toda la imagen contiene información sobre la caja con precintos.

Estas coordenadas las usamos para definir el ROI que se va a poner en la ventana de imagen "Imagen_Mono8".

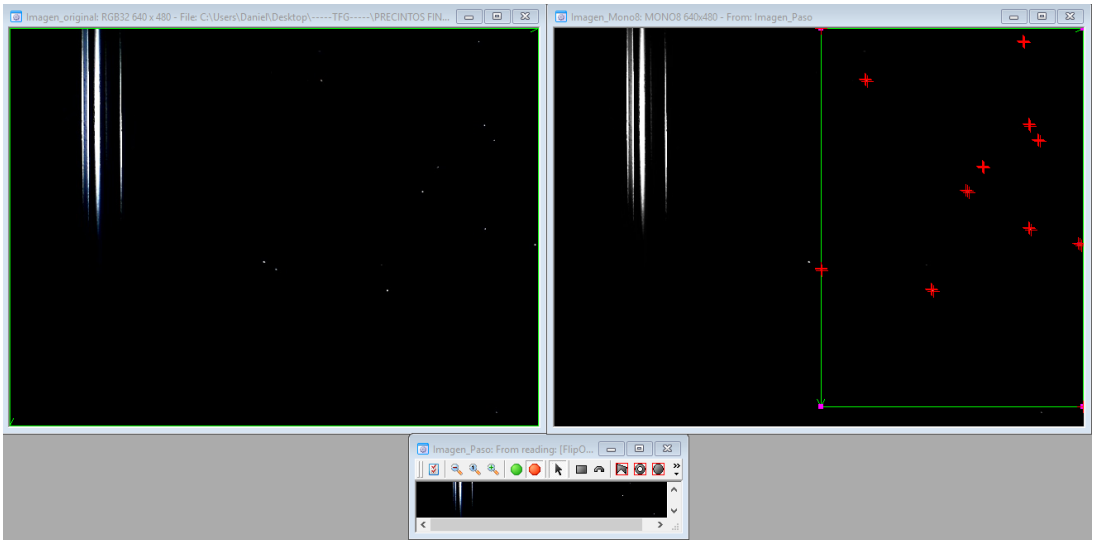


Imagen 41: Ventanas de imagen para el procesado de la imagen de la caja de precintos

En la imagen 41 se ven las ventanas de imagen de *Sherlock* utilizadas para procesar la imagen tomada de la caja de precintos. La imagen grande de la izquierda, "Imagen_original" es la imagen original invertida en X e Y, abajo centrada hay una ventana de imagen, "Imagen_Paso" que tiene la misma imagen que arriba derecha, pero como imagen cargada, y arriba derecha en grande la ventana donde se trabaja realmente, "Imagen_Mono8".

El motivo de necesitar 3 imágenes es el siguiente: la imagen original es RGB, es decir, es una imagen a color, y al cargar imágenes desde un directorio no se pueden modificar de forma previa a la carga. Para los procesados que se realizan después es necesario que la imagen esté en escala de grises, es decir mono8.

Por la posición de la cámara y el movimiento de los motores, la imagen está invertida, en cuanto a que el movimiento positivo en X en la imagen supone un movimiento negativo en la X del motor, y lo mismo para el eje Y. La imagen se carga en color en "Imagen_original", se genera un ROI que ocupa la imagen completa y se aplica una inversión en X e Y. Después se utiliza el algoritmo "Image extract", que hace que el contenido del ROI se pueda utilizar como la imagen a cargar para una ventana de imagen.

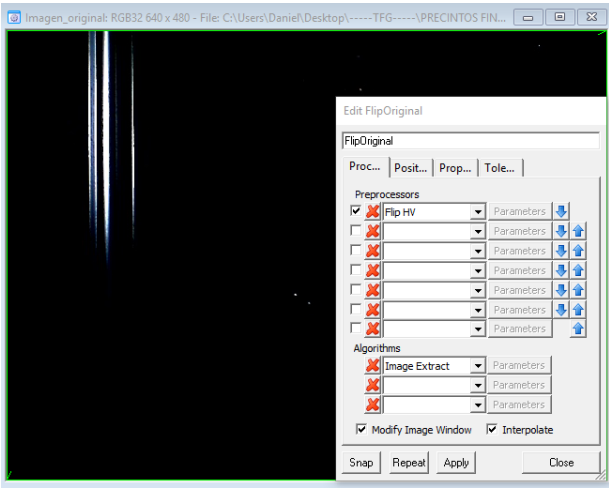
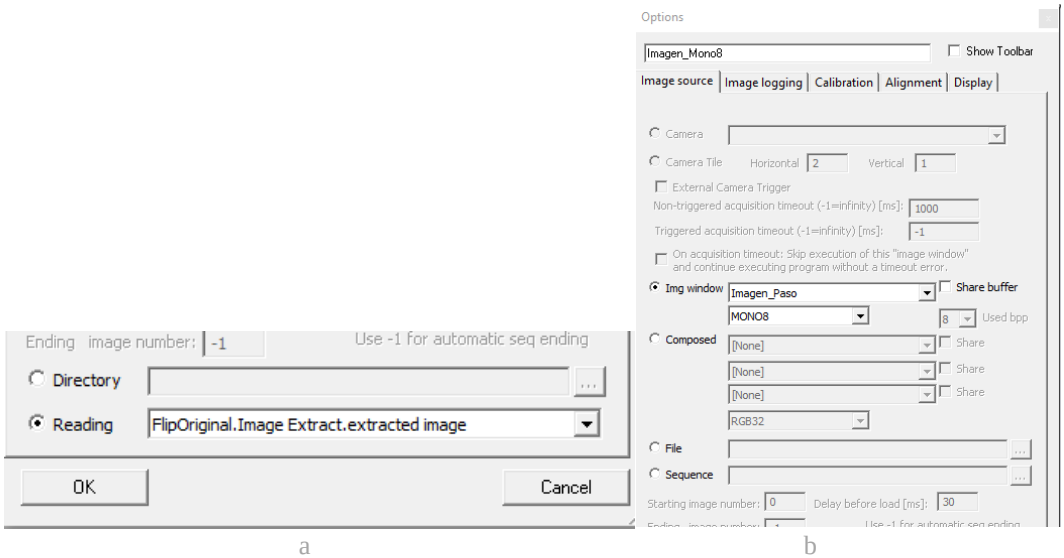


Imagen 42: Configuración del ROI de la ventana de imagen "Imagen_original"

La imagen cargada desde un "Image extract" tampoco se puede modificar en formato, pero si se puede cargar el contenido de una ventana de imagen en otra y modificar el formato a mono8, por eso se carga en "Imagen_Paso" la imagen invertida de "Imagen_original", para poder cargar esta imagen, que es la información directa de la ventana de imagen, no del ROI en la ventana, en "Imagen_Mono8".



a b

Imagen 43: Imagen cargada en ventana "Imagen_Paso" (a). Imagen cargada en "Imagen_Mono8" (b)

Ahora que la imagen está en escala de grises, e invertida para que coincidan los sentidos de los ejes, se genera un ROI en "Imagen_Mono8" y se le introducen como coordenadas las variables que se han modificado en "Selección_ROI_Punto". En la imagen 44 se ve el ROI (Recuadro verde) para la primera posición de la lista

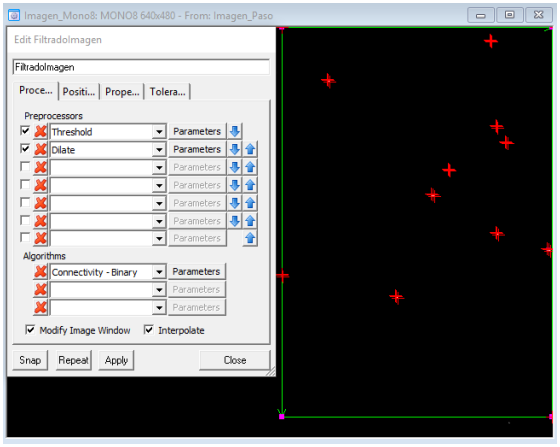


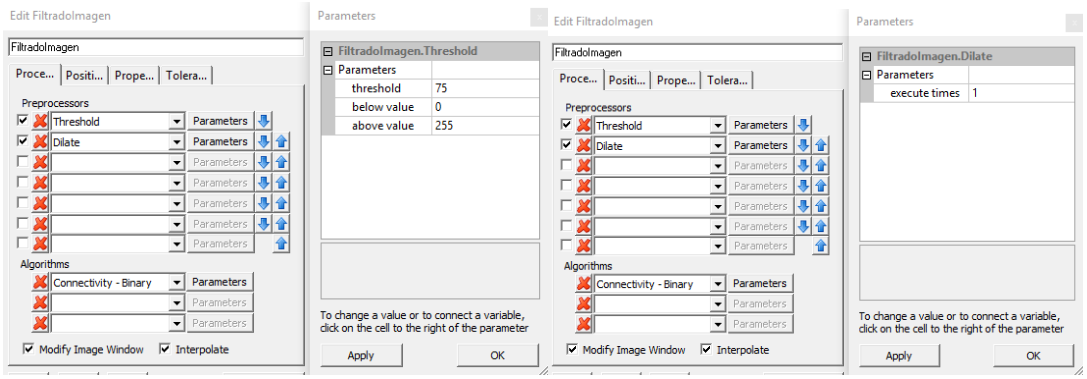
Imagen 44: Configuración del ROI de la ventana de imagen "Imagen_Mono8"

El objetivo en este ROI es utilizar un algoritmo de *Sherlock* llamado "Connectivity-Binary". Este algoritmo detecta *blobs*, blancos o negros según se configure, y extrae su información. Un *blob* es una acumulación de pixeles que estén en contacto con al menos otro píxel de la acumulación. Para poder utilizar este algoritmo no nos basta con que el contenido del ROI esté en escala de grises, necesita estar en blanco y negro.

Para conseguirlo aplicaremos un preprocesado "Threshold", umbralizar en español. El "Threshold" básico, que es el que vamos a utilizar, coge todos los pixeles por encima de un valor en la escala de grises y los pone en el valor que se le indique, y lo mismo para los valores que se encuentran por debajo. Si se le introduce que estos valores que va a aplicar a los pixeles sean 0 y 255, se convierte el contenido del ROI a blanco y negro, pues esos dos valores se corresponden al negro y al blanco en la escala de grises, respectivamente.

El valor de umbral que se le aplica es la variable que se modifica para obtener los resultados deseados, y solo se puede obtener a base de iterar, hasta que al aplicar "Threshold" se obtengan los resultados deseados. El valor que se ha utilizado es 75, lo cual supone que la mayor parte de los valores posibles de píxel se van a transformar a 255, blanco, y se van a considerar un brillo del precinto.

Después del umbralizado aplicamos un preprocesado "Dilate". "Dilate" en imágenes binarias (blanco y negro) coge una matriz 3x3 centrada en cada píxel blanco, y cambia el color de todos los pixeles de esa matriz a blanco. Aplicamos este preprocesado con dos objetivos: eliminar posibles agujeros de 1 píxel que podrían provocar que el brillo de 1 precinto generase 2 *blobs* al dividirse, y para aumentar el área de los *blobs*, para obtener un mejor centro, ya que por la configuración utilizada de la cámara los brillos visibles son muy pequeños, y la parte metálica del precinto es bastante más grande que el brillo que se ve. Solo ejecutamos "Dilate" una vez, porque fue suficiente para obtener buenos resultados.



a b

Imagen 45: Configuración preprocesado "Threshold" en ROI "FiltradoImagen (a). Configuración preprocesado "Dilate" en ROI "FiltradoImagen (b)

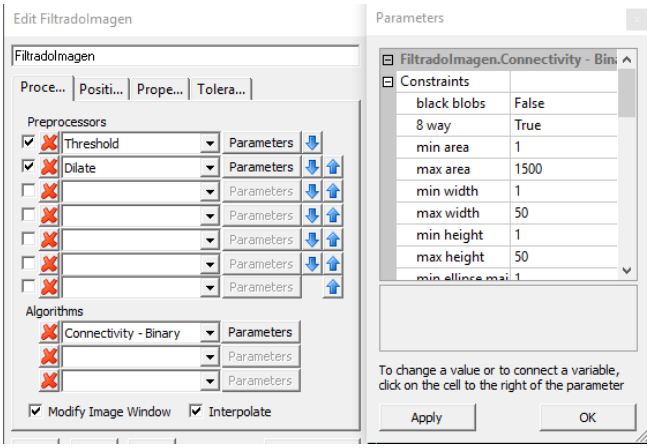


Imagen 46: Configuración algoritmo "Connectivity-Binary" en ROI "FiltradoImagen"

El algoritmo "Connectivity-Binary" se aplica sin restringir la detección de *blobs* más de lo que viene predeterminado, ya que hemos tomado y preprocesado la imagen con el objetivo de utilizar este algoritmo, así que no hace falta filtrar falsos positivos utilizando la configuración de "Connectivity-Binary", y los márgenes predeterminados son más que

suficientes para evitar falsos negativos. Lo que si es necesario configurar es que detecte *blobs* blancos.

De "Connectivity-Binary" se pueden extraer una serie de datos relacionados con los *blobs* detectados. Se van a utilizar dos datos: el número de *blobs* detectados y el centro del *blob* más grande. Se utiliza el *blob* más grande porque el orden predeterminado en el cual "Connectivity-Binary" forma el vector de centros de *blobs* es por tamaño. Ambos datos se van a obtener de la lista de centroides, que se guarda en la variable "CentVect". El número de *blobs* detectados se obtiene comprobando la longitud del vector, y el centroide del *blob* más grande se va a extraer de la posición 0 del vector.

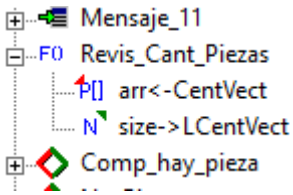
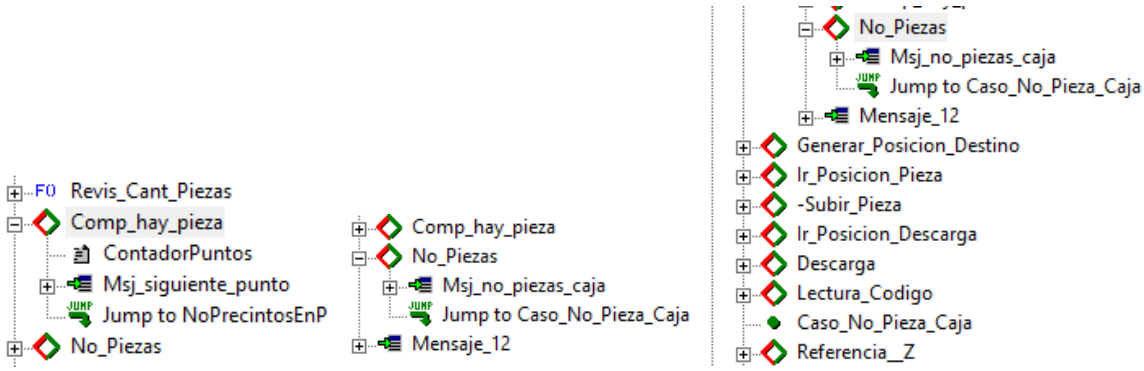


Imagen 47: Función extracción cantidad de precintos detectados

La longitud del vector se guarda en la variable "LCentVect", y se va a comprobar que su valor sea mayor que 0 para continuar en los "IF" "Comp_hay_pieza" y "No_Piezas". Ambos "IF" tienen como condición que "LCentVect" sea igual a 0 para entrar, pero también comprueban el valor de "NPuntosRecorridos". Es en este punto donde se comprueba si es necesario tomar una foto nueva desde otra posición. En caso de que "NPuntosRecorridos" sea menor que 5 se entra en "Comp_hay_pieza", donde se aumenta la cuenta de "NPuntosRecorridos" en 1, en el *script* "ContadorPuntos" y se hace un "Jump" a "NoPrecintosEnP", que se encuentra justo antes de "Ir_Posicion_Foto". En caso de no encontrar un precinto tras pasar por las 6 posiciones marcadas, "NPuntosRecorridos" llega a 5, y se entra en "No_Piezas", donde se hace el "Jump" a "Caso_No_Pieza_Caja", que se encuentra justo antes de "Referencia__Z" al final del bucle de ejecución, lo cual tras la vuelta a referencia de los ejes devuelve el programa al punto de espera de solicitud. Se envía un mensaje por el "*Reporter*" para avisar de lo sucedido.



a b c

Imagen 48: Tarea "Comp_hay_pieza" desplegada (a). Tarea "No_Piezas" desplegada (b). Posición en código del salto a "Caso_No_Pieza_Caja"

En caso de que, si se encuentre un precinto, es decir, "LCentVect" es mayor que 0, se pasa a "Generar_Posicion_Destino", donde se hacen los cálculos necesarios para pasar de pixeles a las unidades de los motores X e Y, y se generan los mensajes que se enviaran a los motores para configurar la posición de P0, que es el punto de recogida.

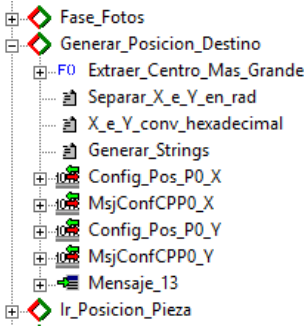


Imagen 49: Tarea "Generar_Posición_Destino" desplegada

Primero se extrae el centroide del *blob* más grande, que es el primero del vector por como esta ordenado, que se guarda en la variable de tipo punto "PosicionMetal", y después se ejecutan 3 *script*s que dan como resultado el mensaje de configuración para los motores.

El *script* "Separar_X_e_Y_en_rad" pasa los valores obtenidos de posición en pixeles a revoluciones para el eje X, y cm para el eje Y.

El *script* "X_e_Y_conv_hexadecimal" pasa los valores obtenidos en el anterior *script* a hexadecimal.

El *script* "Generar_Strings" concatena los valores en hexadecimal obtenidos y las partes fijas del mensaje para crear el mensaje completo necesario.

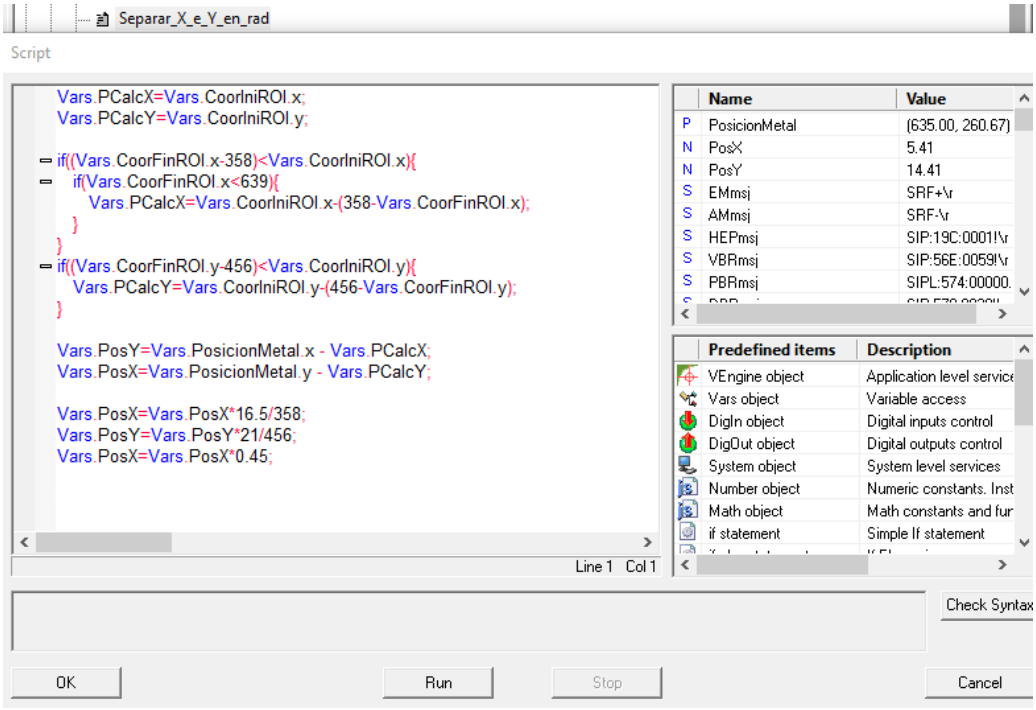


Imagen 50: Código *script* "Separa_X_e_Y_en_rad"

Para poder pasar el valor obtenido en pixeles a las unidades necesarias habrá que aplicar un factor de conversión, pero antes de hacerlo se modificarán los valores en pixeles, ya que están contados desde la esquina superior izquierda de la imagen, y no tiene por qué coincidir con la esquina de la caja en la imagen. Las coordenadas que definen el ROI si utilizan la esquina de la caja, están referenciadas a esta, o se puede obtener la esquina de la caja utilizando estos datos, dependiendo de a qué posición haya ido el robot a tomar la foto.

En caso de ser visible la esquina superior izquierda de la caja en la imagen, habrá que restar al valor del centroide obtenido la posición de la esquina superior izquierda del ROI. Al hacer esto se referencia el punto obtenido a la esquina de la caja y ya se puede realizar la conversión de unidades. Cuando este no es el caso, son necesarios más cálculos.

- Primero se guarda el valor de las coordenadas X e Y de la esquina superior izquierda del ROI en las variables "PCalcX" y "PCalcY".
- Segundo se comprueba si entre el valor inicial y el final en X del ROI hay 358 pixeles, que es el ancho de la caja pasado a pixeles. Esto se comprueba restando al valor final 358, y comprobando si es menor que el inicial. De cumplirse que es

menor significa que el ROI no contiene la caja entera, y podría no verse la esquina superior izquierda, así que se entra en el "IF".

Dentro del "IF", para comprobar si no se ve esta esquina, se revisa si el valor final de la X del ROI es inferior a 639.

Se usa este valor porque la imagen es de 640x480 pixeles.

- Si el valor es inferior significa que la parte que no se ve de la caja queda a la izquierda de la imagen, y no se ve la esquina superior izquierda.
- Si el valor no es inferior significa que la caja queda a la derecha, y si se ve la esquina.

En el caso de ser inferior se entra en el "IF", y se modifica el valor guardado en "PCalcX". El nuevo valor será la diferencia entre el ancho del ROI y el ancho de la caja en pixeles, en negativo. Con este cálculo se obtiene la posición de la esquina superior izquierda cuando queda fuera de la imagen, ya que tiene como resultado cuantos pixeles se quedan fuera de la imagen.

- Tercero se comprueba, de la misma manera que para la X, si entre el valor inicial y final de la Y hay una diferencia del alto en pixeles de la caja, que es 456.
  - En caso de que la diferencia sea menor significa que parte de la caja no se ve., pero a diferencia del caso de la X, por como son los puntos a los cuales va el robot a colocar la cámara, solo se puede perder parte de la caja por arriba.

Cuando es inferior se entra en el "IF" y se modifica el valor de "PCalcY" por la diferencia entre la altura del ROI y la altura de la caja en pixeles, en negativo.

Con los valores de "PCalcX" y "PCalcY" ajustados según la posición de la caja en la imagen, se sustraen estos valores a los del centroide del *blob* más grande, se guardan en las variables "PosX" y "PosY", y se aplica el factor de conversión a estos valores.

Para el eje X se obtiene dividiendo 16,5 entre 358, que son las dimensiones alcanzables de la caja de precintos en centímetros y en pixeles, respectivamente, y luego multiplicando el resultado por 0,45 para pasar el resultado a revoluciones. Este valor se

obtuvo realizando pruebas con el motor, dando órdenes de x revoluciones y comprobando cuanto se desplazaba el eje en cm.

Para el eje Y se repite el mismo proceso, con los valores 21 y 456, pero no es necesario un segundo factor pues la entrada del eje Y es directamente en cm.

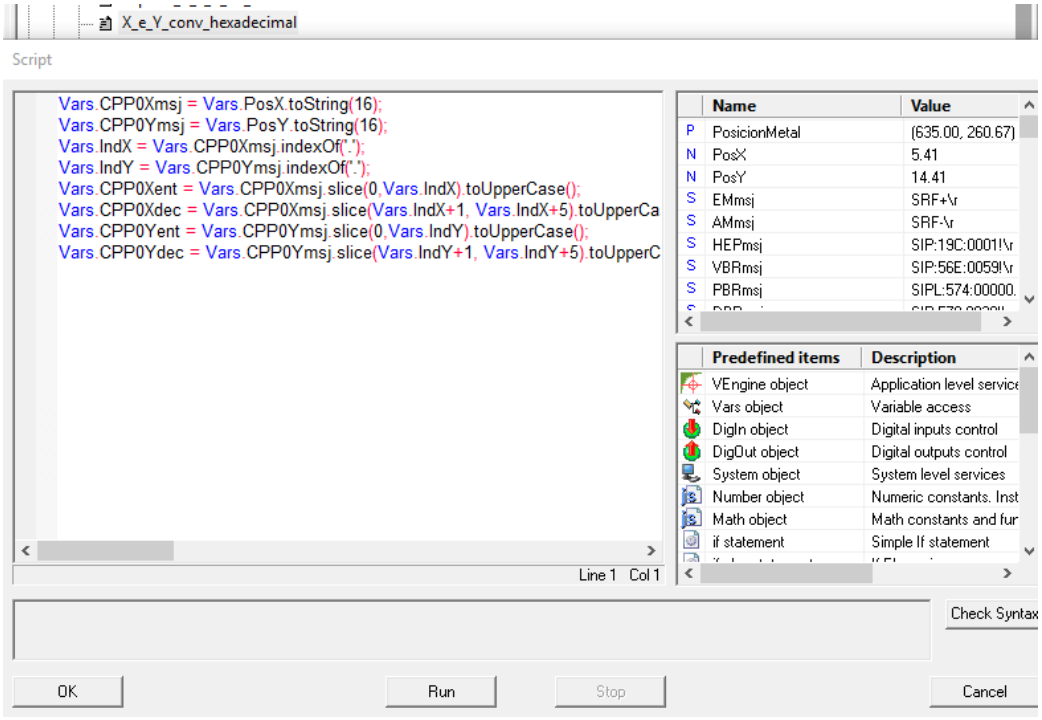


Imagen 51: Código *script* "X_e_Y_conv_hexadecimal"

Una vez se tienen los valores de las posiciones a las que hay que mover los motores, se preparan para generar los mensajes a enviar. Los mensajes contienen 8 cifras en hexadecimal, las 4 primeras son los valores enteros y las 4 últimas los decimales.

El mensaje a enviar tiene que ser una variable de tipo "string", y ahora mismo los valores de las posiciones están guardados en variables de tipo "número". La conversión del tipo de variable se hace lo primero para poder utilizar las funciones de *Javascript* que interactúan con *strings*.

Para hacer la conversión se utiliza la función "toString()", que al convertir un número a texto da la opción de cambiar el formato, y en nuestro caso necesitamos cambiar los valores de formato decimal a formato hexadecimal. Para esto se pone en el paréntesis de la función "toString()" un 16, que le indica el nuevo formato deseado.

Guardamos los valores convertidos a *string* en las variables "CPP0Xmsj" y "CPP0Ymsj".

Lo siguiente que se hace es localizar en el *string,* que es un vector de símbolos, la posición del símbolo ".", que es la coma que separa los decimales. Esto se hace para poder separar en dos partes el número, una con el valor entero y otra con el decimal.

El símbolo "." se localiza con la función "indexOf()", a la cual se le introduce el símbolo "." como entrada, y se guarda su índice en las variables "IndX" e "IndY".

Con la posición guardada de "." separamos el número en dos partes, entero y decimal, con la función "slice". Los 4 números resultantes se van a guardar en las variables "CPP0Xent", "CPP0Xdec", "CPP0Yent" y "CPP0Ydec".

La función "slice" necesita dos entradas, el índice del símbolo en el cual se quiere empezar el recorte del *string,* y el índice del símbolo final. La función "slice" incluye en el recorte los símbolos del *string* que se encuentran entre ambos valores, e incluye en el recorte el símbolo del valor inicial. Con esto en mente, las entradas para la función "slice" se definen de la siguiente manera:

Para las partes enteras del número, se utiliza como primera entrada 0 y como segunda entrada el índice del símbolo ".".

Para las partes decimales del número, se utiliza como primera entrada el índice del símbolo "." + 1, y como segunda entrada el índice del símbolo "." + 5, para que solo coja 4 decimales.

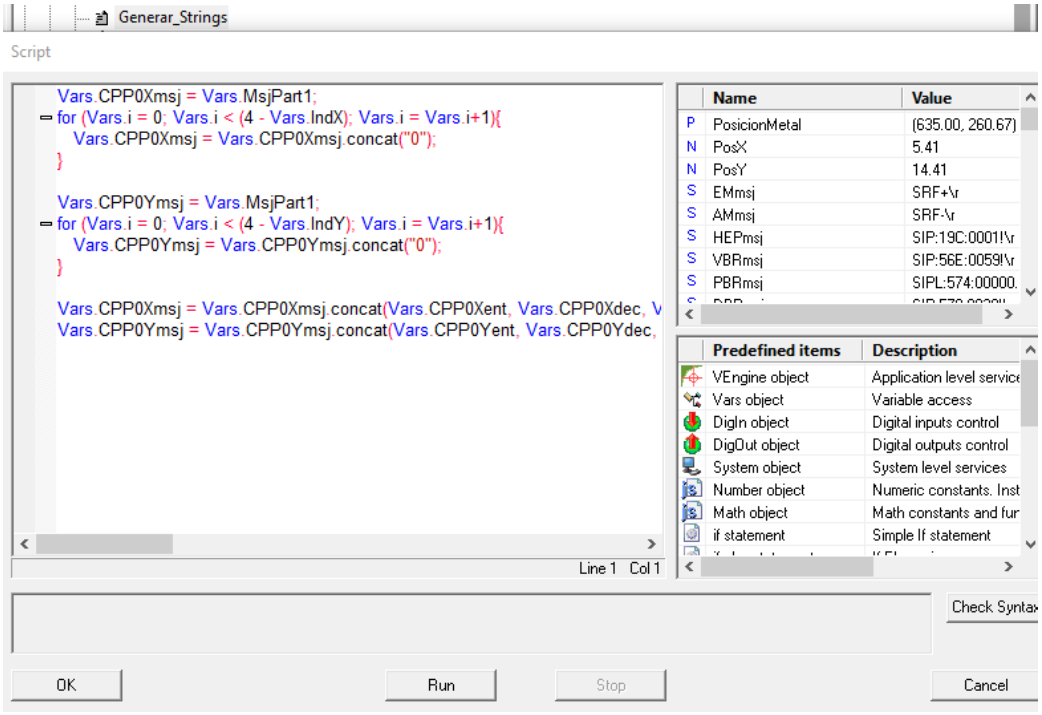


Imagen 52: Código *script* "Generar_Strings"

Con ambos valores de las posiciones en hexadecimal, y separados por parte entera y decimal, el programa entra en el *script* "Generar_Strings", y monta, utilizando las diferentes partes preparadas, el mensaje.

El mensaje tiene una parte y una parte variable, que es el valor de la posición. La parte fija del mensaje se ha preparado en las variables de tipo *string* "MsjPart1" y "MsjPart2". La parte fija del mensaje está separada en dos trozos para facilitar el concatenado.

El mensaje tiene el siguiente aspecto: SIPL:590:XXXXXXXX!\r. Las 8 "X" son el espacio donde va la posición, y lo demás es fijo. Al haber parte delante, y parte detrás, se introduce ese texto en dos variables distintas y después se concatena el mensaje completo.

El espacio para la posición es de 4 valores para la parte entera y 4 para la decimal, por eso en el anterior *script* se recogían solo 4 decimales del dato. El problema que queda por tratar antes de concatenar es que la parte entera podría no tener 4 valores, y el mensaje requiere de 4 cifras para la parte entera.

Para solucionarlo se iguala el valor de la variable que se va a usar para el mensaje completo, “CPP0Xmsj” o “CPP0Ymsj”, a “MsjPart1”, y se concatenan tantos ceros a ese mensaje como sean necesarios para cubrir el hueco que deja el valor entero. Esto se hace con un bucle “for” que ejecuta concatenar 1 cero al mensaje tantas veces como la diferencia entre 4 y el índice de la coma del decimal, que se guardaron previamente en las variables “IndX” e “IndY”.

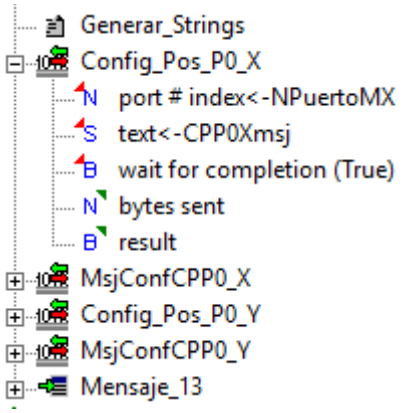


Imagen 53: Comunicaciones configuración posición punto P0

Una vez añadidos los ceros necesarios se concatenan al mensaje el contenido de las variables que contienen la parte entera (“CPP0Xent” o “CPP0Yent”), la parte decimal (“CPP0Xdec” o “CPP0Ydec”) y la variable “MsjPart2”, en ese orden.

Por último, se envían los mensajes preparados a los motores para que se configure la nueva posición del punto “P0”.

Una vez enviada la configuración de “P0” a los motores se terminan los cálculos y configuraciones previas a la recogida de un precinto.

Lo siguiente que realiza el programa son los movimientos de recogida y descarga. El brazo se moverá a "P0", bajará el eje z, lo subirá al coger un precinto, el brazo se moverá a "P2" y el eje Z bajará al punto de descarga.

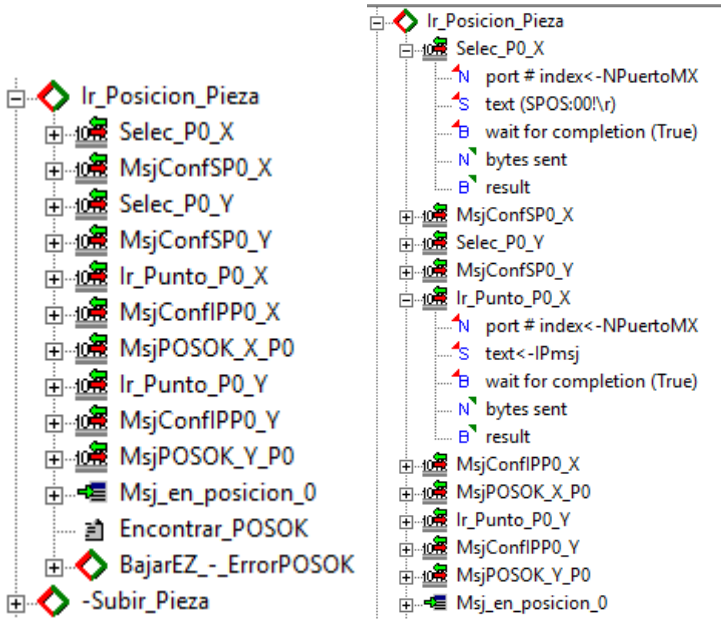


Imagen 54: Tarea "Ir_Posicion_Pieza" desplegada (a). Comunicaciones movimiento a P0 (b)

En el primer paso de la recogida y descarga se realiza la tarea "Ir_Posicion_Pieza", donde se realizan las comunicaciones de selección de punto, "P0" para el punto de recogida, y de marcha al punto. En este movimiento de los ejes X e Y si se comprueba el mensaje de confirmación de llegada al punto de destino, ya que se va a mover el eje Z.

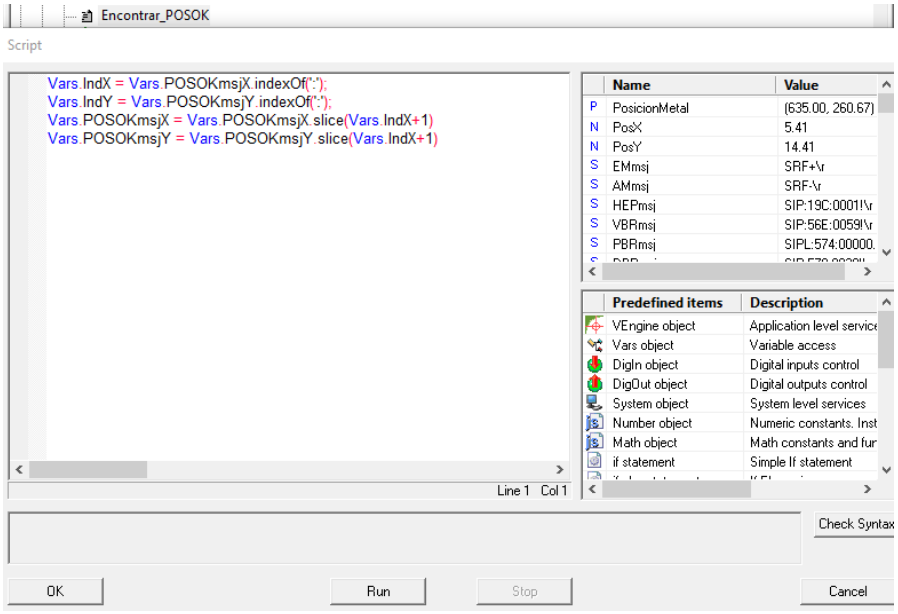
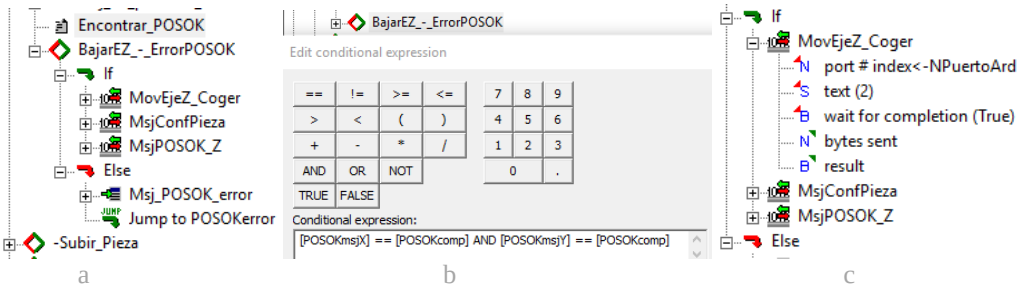


Imagen 55: Código script "Encontrar_POSOK"

En el *script* "Encontrar_POSOK" se extrae de los mensajes de confirmación de llegada al punto de destino, guardados en las variables "POSOKmsjX" y "POSOKmsjY", el último trozo, que es muy característico, "POSOK", y se utiliza para hacer la comprobación. En este mensaje de confirmación hay un ":" justo antes de "POSOK", así que se utiliza para saber a partir de que índice del *string* se ha de realizar la función "slice" para solo recoger el trozo de interés del mensaje. La función "slice" con solo una entrada recoge todos los datos a partir de la posición que se le introduce, incluyendo la posición introducida.

Los trozos de mensaje extraídos se guardan en las variables "POSOKmsjX" y "POSOKmsjY" y se comprueban en el "If else" "BajarEZ_-_ErrorPOSOK" comparándolos con la variable "POSOKcomp".



a b c

Imagen 56: Función "BajarEZ_-_ErrorPOSOK" desplegada (a). Condición bajar eje Z recogida de precinto(b). Comunicaciones eje Z recogida de precinto (c)

En caso de que la condición no se cumpla se envía un mensaje por el *reporter* y se hace un "Jump" a "POSOKerror", que se encuentra después de la vuelta a referencia del eje Z del bucle de ejecución, lo cual ejecuta la vuelta a referencia de los ejes X e Y, y devuelve el programa al punto de espera de solicitud.

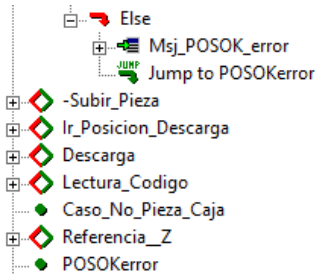
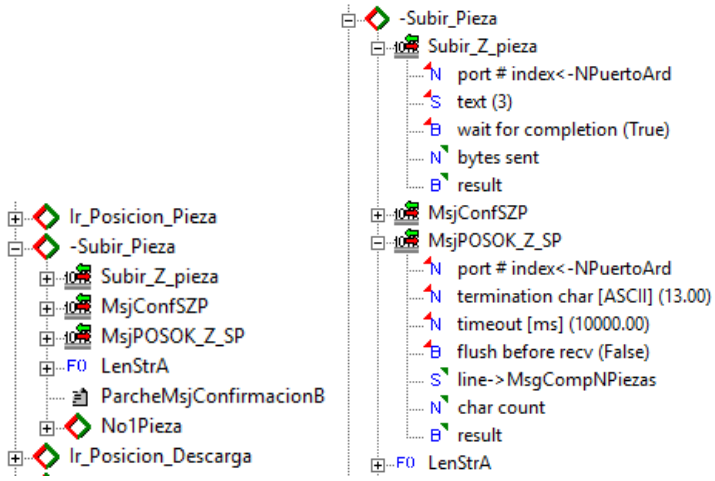


Imagen 57: Posición en código del salto a "POSOKerror"

En caso de que se cumpla la condición de que ambos mensajes sean iguales a la comprobación se envía un "2" al *Arduino* para que baje el eje Z con el imán encendido y comprobando la carga para determinar cuándo parar. Cuando el eje ha bajado hasta recoger un precinto el *Arduino* devuelve un mensaje de comprobación, que se recoge en "MsjPOSOK_Z" y sirve de espera hasta que se complete el proceso.



a b

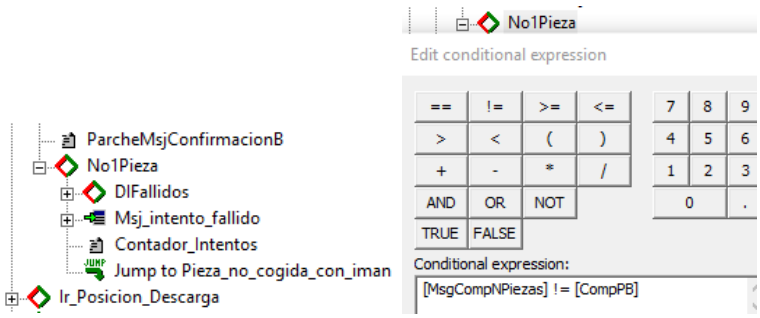
Imagen 58: Tarea "Subir_Pieza" desplegada (a). Comunicaciones subidas de precinto eje Z (b)

Una vez el eje Z ha bajado hasta la caja de precintos y ha parado por contacto el eje tiene que subir, para lo que se envía un "3" al *Arduino* para darle la orden. Al tener el imán encendido y haber bajado justo encima de la pieza de metal de un precinto debería subir con él, pero puede haber complicaciones.

La primera complicación posible es que el precinto esté enterrado, y el eje no puede sacarlo al no tener suficiente fuerza el imán, y la segunda es que al subir el precinto se hayan enganchado más precintos, ya sea a la punta del cabezal por haber estado muy cerca del punto de bajada, aunque no fuesen el objetivo, o por estar enganchados al precinto que se quiere subir.

Durante la subida del eje Z el *Arduino* hace una pequeña pausa para medir el peso que tiene cargado, y comparando con el valor de la tara que realiza en cierto momento, sin carga, y el valor que se extrajo durante el desarrollo de cuanto pesa un precinto, determina cuantos precintos está cargando. Dependiendo de cuantos precintos detecta en ese momento el *Arduino* toma unas acciones u otras. En caso de que detecte un precinto sigue con la subida, pero en caso de que detecte otra cantidad de precintos apaga el electroimán y después termina de subir. Cuando termina de subir envía un mensaje de confirmación que también sirve para comunicar cuantos precintos se habían recogido, 1 o un valor distinto de 1.

El mensaje de confirmación se guarda en la variable "MsjCompNPiezas", y se le aplica el parche para eliminar el salto de línea que aparece sin motivo.



a b

Imagen 59: Tarea "No1Pieza" desplegada (a). Condición tarea "No1Pieza" (b)

Con el mensaje de comprobación arreglado se entra en el "IF" "No1Pieza", donde se compara si "MsjCompNpiezas" es distinto de la variable de comprobación "CompPB". "CompPB" contiene el mensaje esperado en caso de que se haya recogido 1 precinto, con lo que a "No1Pieza" se entra solo si se recoge una cantidad distinta de 1 precinto.

Cuando se cumple que "MsjCompNpiezas" es distinto de la variable de comprobación "CompPB", y se entra en "No1Pieza", se comprueba la cantidad de veces que esto ha ocurrido desde la solicitud de precinto. La variable contador utilizada es "NIntentos", y en el "IF" "DIFallidos" se comprueba si ha llegado a 10. Este valor es arbitrario, podría ser cualquier otro, y se comprueba para que el programa no se quede atrapado tratando de recoger el mismo precinto que está enterrado y no llega a moverse nunca al intentar recogerlo. En caso de que no haya llegado a 10 se aumenta la cuenta en 1 en el *script* "Contador_Intentos" y se hace un "Jump" a "Pieza_no_cogida_con_iman", que se encuentra justo antes de "Fase_Fotos", y hace que se repita el proceso de toma de imagen

y selección de punto al que ir a recoger. Se vuelve a tomar una imagen porque al bajar el eje Z se pueden haber movido de posición los precintos.

Si "NIntentos" ha llegado a 10 se entra en "DIFallidos", donde se hace un "Jump" a "Caso_No_Pieza_Caja", que se encuentra antes de la vuelta a referencia del eje Z del bucle de ejecución, lo cual hace que se ejecuten todas las vueltas a referencia y se vuelva a la espera de solicitud de precinto.

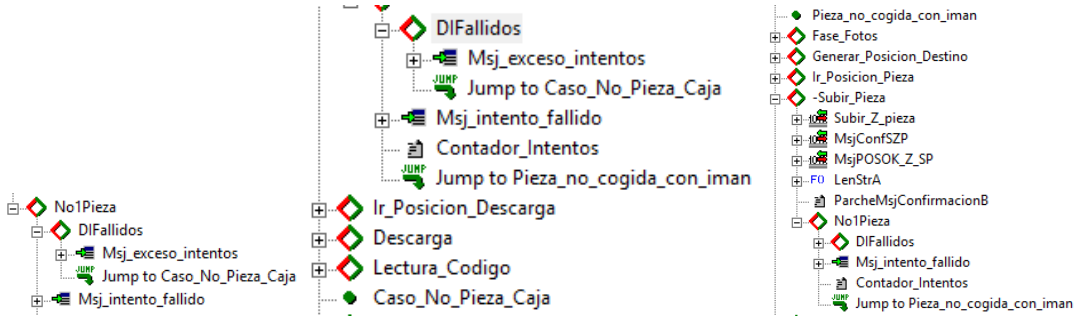


Imagen 60: Tarea "No1Pieza" desplegada(a). Posición en código del salto a "Caso_No_Pieza_Caja" (b). Posición en código del salto a "Pieza_no_cogida_con_iman" (c)

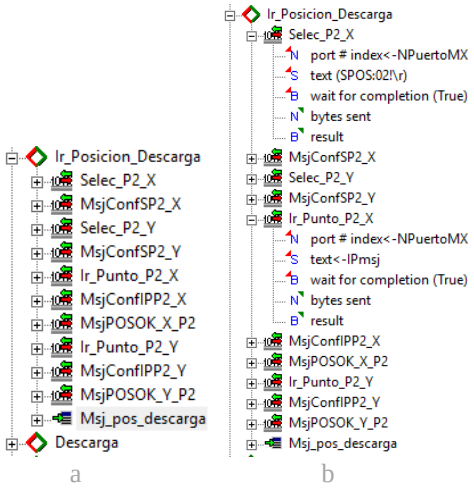
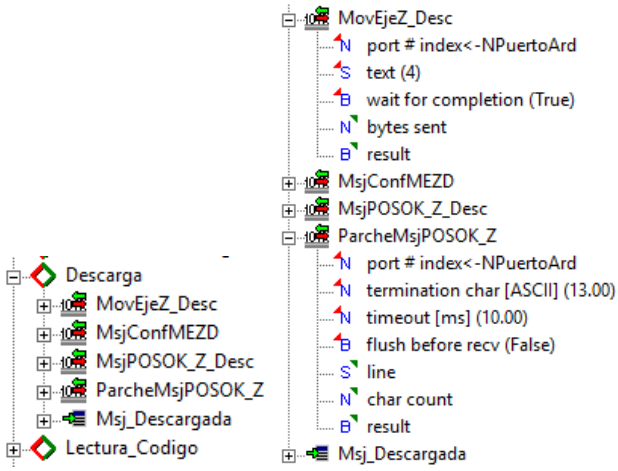


Imagen 61: Tarea "Ir_Posicion_Descarga" desplegada(a). Comunicaciones movimiento a punto P2 (b)

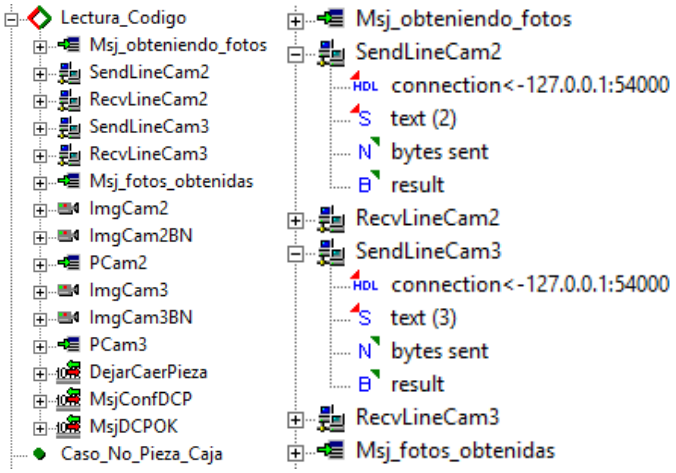
En caso de que no se entre en "No1Pieza", ya que se ha recogido 1 precinto, se continua el proceso moviendo los ejes X e Y hasta "P2", que es el punto de descarga.



a b

Imagen 62: Tarea "Descarga" desplegada (a). Comunicaciones descarga (b)

Una vez en "P2" se envía un "4", que es la orden de descarga. En la orden de descarga el eje Z baja una cantidad especifica que deja el precinto a 1cm de la base del bloque de captura, y una vez está ahí apaga el electroimán para dejar caer el precinto hasta la posición en la que se toman las imágenes para el análisis del código del precinto.



a b

Imagen 63: Tarea "Lectura_Codigo" desplegada (a). Comunicaciones toma imágenes precinto (b)

Con el precinto en posición se entra en la tarea "Lectura_Codigo", donde se envían los mensajes para tomar las imágenes del código y se analizan. Se envían los mensajes "2" y "3" al programa en *Python* para que tome las imágenes de las cámaras del bloque de captura

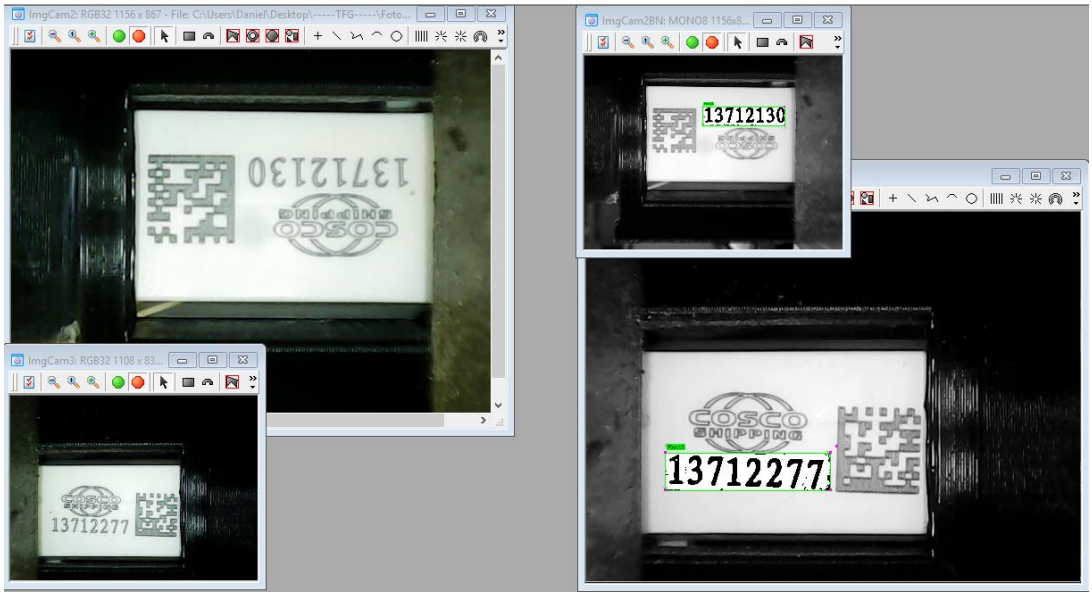


Imagen 64: Ventanas de imagen para la lectura del código del precinto

En la imagen 64 se ve la pantalla de *Sherlock* con las cuatro ventanas de imagen que se utilizan para el análisis del código del precinto. En las dos ventanas de la izquierda se cargan ambas imágenes tomadas, y no se modifican. Se cargan las imágenes en ventanas de imagen por el mismo motivo que cuando se analiza la imagen del recipiente de precintos, para poder cargar después la misma imagen en escala de grises, mono8, y poder utilizar ciertos algoritmos de *Sherlock*. Las ventanas de imagen de la derecha contienen las imágenes cargadas en mono8.

Por cómo es necesario instalar la cámara izquierda del bloque de captura, la asociada al "2" en el programa del *Sherlock*, la imagen queda girada 180º. Para evitar el paso extra que supone extraer de un ROI en la imagen original la imagen invertida, se invierte después solo la zona de interés en la imagen en mono8.

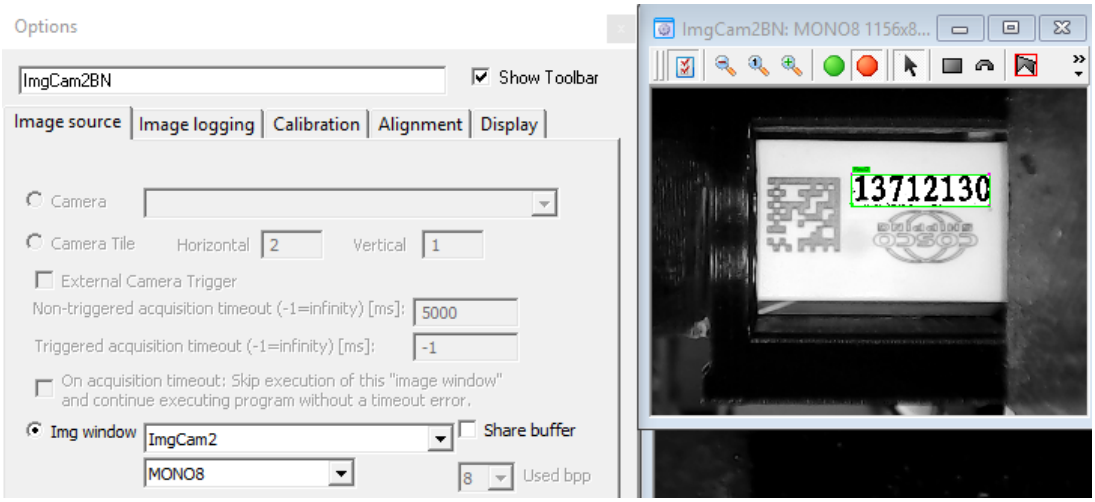
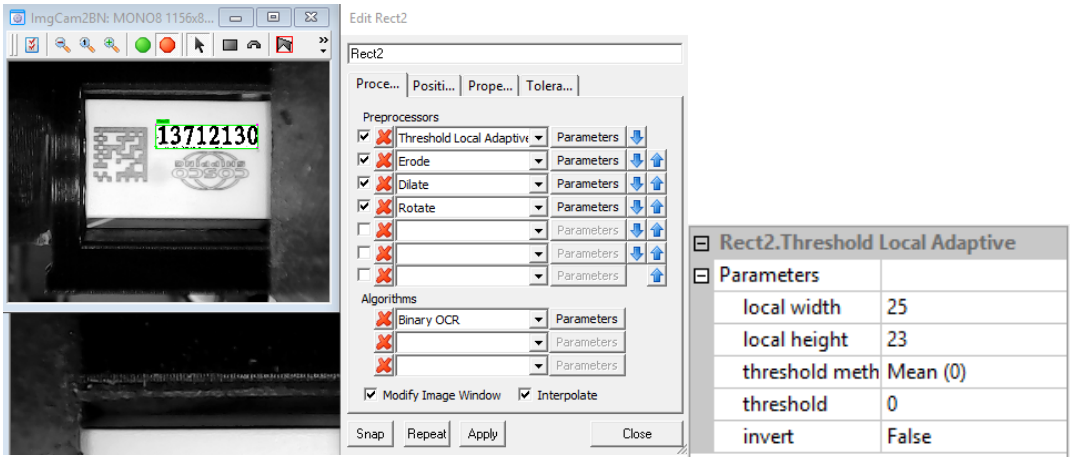


Imagen 65: Imagen cargada en ventana "ImgCam2BN"

Los ROI de las imágenes en mono8 se colocan de forma que solo cogen el código numérico, con un poco de margen ajustado probando distintas imágenes, para asegurar que el número nunca se queda fuera, ya que la posición, tanto del precinto como del código en el precinto, no son siempre exactamente iguales.



a b

Imagen 66: Configuración del ROI de la ventana de imagen "ImgCam2BN" (a). Configuración preprocesado "Threshold Local Adaptative" (b)

En la imagen 66 se ven los procesados utilizados. El algoritmo que se utiliza para extraer los números de la imagen es "Binary OCR". "Binary OCR" requiere que el contenido del ROI en el que se aplica esté en blanco y negro, no escala de grises, con lo que es necesario aplicar un umbralizado.

El umbralizado que se utiliza en estas imágenes es “Threshol Local Adaptative”. Este tipo de umbralizado se utiliza en imágenes que no tienen un nivel de contraste constante, y en los precintos, al existir la posibilidad de que no fuesen perfectamente perpendiculares a la cámara y fuente de luz, por el margen dado en el bloque de captura para que no se atascasen, aparecen diferentes niveles de brillo. Este método de umbralizado asigna un valor de umbral variable dependiendo de los pixeles que rodean a cada píxel procesado, dependiendo de la configuración dada. En la imagen 66b se ven los valores asignados a la configuración, y el método escogido, que fueron escogidos por iteración.

Después del umbralizado se aplica un “Dilate”, ya explicado durante el preprocesado de la imagen del recipiente de precintos, y un “Erode”, que es un procesado contrario a “Dilate”. “Erode” coge una matriz 3x3 centrada en cada píxel negro, y cambia el color de todos los pixeles de esa matriz a negro. Aplicar estos dos preprocesados juntos tiene como finalidad eliminar pixeles sueltos que actúen como ruido, sin modificar el tamaño de los *blobs*, ya que el borde de cada *blob* pierde 1 píxel en el “Dilate” y gana 1 píxel en el “Erode”. Ambos preprocesados se ejecutan 1 vez.

El preprocesado “Rotate” que se ve en la imagen 66a gira en un ángulo especificado el contenido del ROI. Se utiliza en la imagen de la cámara izquierda para corregir el problema de la instalación de la cámara, para no tener que tratar los datos que salgan del algoritmo “Binary OCR”. En la imagen de la cámara derecha no se utiliza, el resto del preprocesado es idéntico.

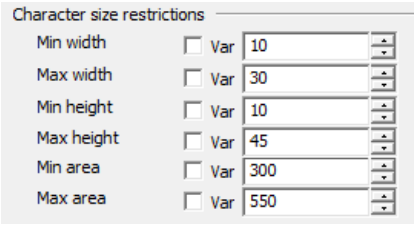


Imagen 67: Configuración restricciones algoritmo “Binary OCR”

Con la imagen preprocesada se aplica el algoritmo “Binary OCR”. Este algoritmo detecta los *blobs* del ROI que entran en los márgenes designados, los ordena, y en la ventana de entrenamiento permite al usuario introducir que símbolo asignar a cada *blob*. Al signar los símbolos, el algoritmo aprende la forma del *blob*, y la próxima vez que detecte *blobs* similares les asigna el mismo símbolo. Para entrenar bien el algoritmo es necesario introducir múltiples ejemplos.

Los márgenes de detección de *blobs* se pueden ver en la imagen 67, que se han seleccionado por iteración.

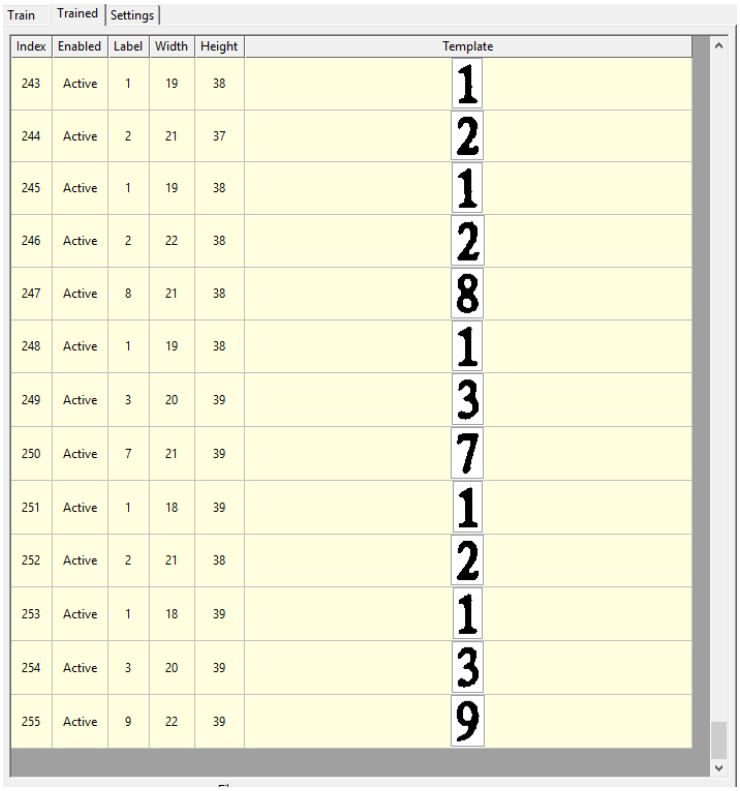


Imagen 68: Ejemplo números entrenados en la función "Binary OCR"

En la imagen 68 se puede ver un ejemplo del aspecto que tiene la lista de símbolos entrenados. La lista completa no se muestra porque contiene 255 muestras.

Del algoritmo se puede extraer el *string* que contiene la lista de símbolos detectados. Se guardan estos *string* en las variables "Cam2String" y "Cam3String", y se muestran por el *reporter*.

Después de procesar las imágenes se envía un "8" al *Arduino*, que abre y cierra la trampilla del bloque de captura, dejando caer el precinto procesado.

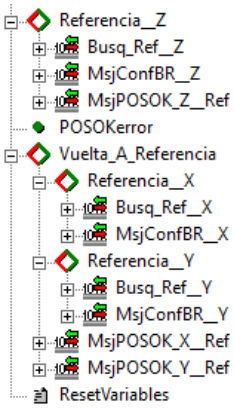


Imagen 69: Tarea "Vuelta_A_Referencia" desplegada

Con el código extraído, se envían las ordenes de vuelta a referencia a los motores, se reinician las variables, y el programa vuelve a esperar la solicitud de un precinto.

En el caso de que sea el último precinto que se solicita en la sesión, se pulsa espacio durante la espera de solicitud de un precinto para apagar el programa. Pulsar espacio provoca un "Jump" a "ProgEnd", que apaga los motores y finaliza el programa.

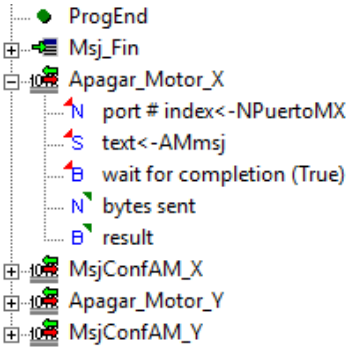
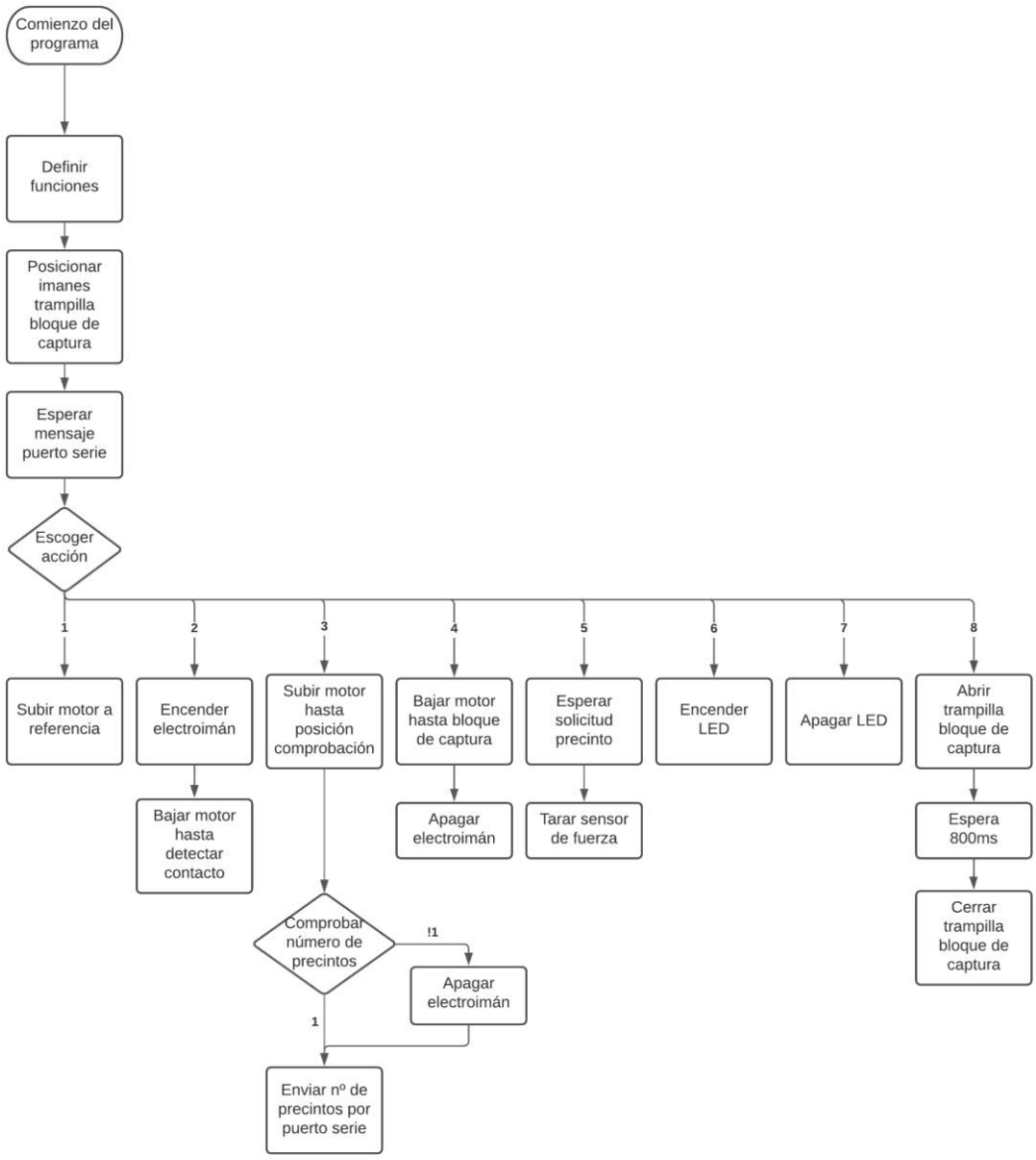


Imagen 70: Comunicaciones apagado motor

Programa *Arduino*



Flujograma 4: Programa *Arduino*

Este programa controla el uso del *Arduino* como tarjeta de entradas y salidas para el ordenador, y el motor del eje Z.

Para realizar esas tareas se definen dos funciones, y se diseña un programa que responde a los distintos mensajes que llegan por el puerto serie desde el programa principal en *Sherlock*.

Las conexiones del *Arduino* se pueden ver en el anexo 4.

La primera función que se define, y la que sirve de corazón para el control del motor del eje Z, es la función "step".

```
void step(boolean dir, byte dirPin, byte stepperPin)

{
  digitalWrite(dirPin, dir);
  digitalWrite(stepperPin, HIGH);
  delayMicroseconds((int)delayTime);
  digitalWrite(stepperPin, LOW);
  delayMicroseconds((int)delayTime);

  if(dir==1)
  {
    valorStep=valorStep+1;        //CONTADOR PARA SABER EL NÚMERO DE PASOS
  }
  else
  {
    valorStep=valorStep-1;
  }
}
```

Imagen 71: Código función "Step"

Esta función tiene 3 entradas: una dirección que se introduce como "True" o "False", el pin por el cual se comunica la dirección al driver del motor paso a paso, el pin de entrada de la señal de control al driver.

La función "step" primero envía, por el pin de dirección, el valor de la dirección que se le introduce, y después genera un pulso cuadrado de duración igual a la variable "delayTime" utilizando esperas. La variable "delayTime" no se va a mantener fija a lo largo del programa, pues modificarla afecta a la velocidad del motor, que reacciona a los flancos de la señal cuadrada que se forma. Cuanto menor es "delayTime", más rápido gira el motor.

Durante el desarrollo se hicieron pruebas para ver cómo de rápido se podía utilizar el motor, pero se vio que con señales de alta frecuencia el motor no era capaz de funcionar al no conseguir reaccionar a tiempo, y girar lo necesario antes de que llegase el siguiente paso. Cuando esto ocurre el motor empieza a saltar pasos y se queda bloqueado. Lo que se hizo, que esta implementado fuera de la función, fue introducir aceleración al motor. Empezando con una señal de frecuencia baja que no da problemas, una vez está girando es capaz de no saltar pasos al pasar a una frecuencia mayor, al tener velocidad previa. Se sube progresivamente la frecuencia hasta alcanzar un valor que no da problemas, más alto que el que se podía conseguir antes.

En el código se hace reduciendo el valor de "delayTime" después de cada paso, hasta que alcanza el valor mínimo que permite que no haya errores, que esta guardado en la variable "delayMin".

```
void gotoStart()
{
  TOP_END=0;
  while (TOP_END==0)
  {
    step(SUBIR, X_DIR, X_STP);
    TOP_END = digitalRead(X_END_TOP);
    if(delayTime>=delayMin){
      delayTime = delayTime - 0.5;
    }
  }
  valorStep=0;
}
```

Imagen 72: Código función "gotoStart"

La otra función que se define es "gotoStart", que es la función que manda el eje Z a referencia, en su posición más alta.

La variable "TOP_END" es la lectura del final de carrera instalado en la parte superior del eje, marcando el final de este. Esta función ejecuta la función step en la dirección de subida, lee la entrada digital del final de carrera para actualizar la variable "TOP_END", y ajusta el valor de "delayTime" mientras se cumple que el final de carrera no está siendo pulsado.

Este movimiento, subir hasta el final de carrera, es el único movimiento definido en una función porque es el único que se utiliza más de una vez en el programa.

```
digitalWrite(IMANODER, HIGH);
digitalWrite(IMANOIZQ, LOW);
delay(200);
digitalWrite(IMANOIZQ, HIGH);
```

Imagen 73: Código preparación trampilla

Aparte de las configuraciones de los pines del *Arduino* y la inicialización de variables, antes de entrar en el bucle de ejecución se ejecutan las 4 líneas de código que aparecen en la imagen 73.

Estas 4 líneas activan y desactivan los solenoides de la trampilla del bloque de captura en una secuencia que la cierra independientemente del estado anterior.

La trampilla se encuentra cerrada cuando está colocada a la izquierda de su rango de desplazamiento, así que lo primero que se hace es apagar el solenoide derecho, por la posibilidad de que se haya quedado encendido, para liberar la trampilla.

Con la trampilla liberada se enciende el solenoide izquierdo, que tira de la trampilla hacia la izquierda, cerrándola, se deja encendido un pequeño espacio de tiempo, 200 milisegundos, para dar tiempo a que se cierre y se apaga el solenoide izquierdo.

Bucle de ejecución

```
dato = 0;
while((dato < 49)or(dato > 56)){
  dato=Serial.read();
  delay(200);
}
```

Imagen 74: Código espera lectura puerto serie

Dentro del bucle el programa espera a recibir una orden del programa principal antes de actuar. Al entrar al bucle de ejecución se deja el programa en un bucle que lee la entrada del puerto serie hasta que el mensaje que se extraiga entre en los definidos como válidos.

La función “read” de la librería de comunicación serie de *Arduino* recoge toda la información guardada en el buffer en el momento de lectura, pero la recoge en valores ASCII. Esto significa que cualquier símbolo que llegue, ya sea letra o número, se extrae por su código en la tabla ASCII. Para evitar complicar la lectura del mensaje, se utilizaron números simples para los mensajes, que son los números del 1 al 8 para las 8 posibles ordenes que puede recibir. En la tabla ASCII el número “1” tiene el valor 49, y los números van en orden ascendente, siendo el valor de “2”, 50, y terminando para nosotros en 56, el valor de “8”.

A la hora de comparar que mensaje ha llegado se utiliza directamente el valor de la tabla ASCII. El bucle se ejecuta mientras la lectura del puerto serie no devuelva un valor aceptado, que van del 49 al 56, ambos incluidos, guardando en cada ejecución el valor en la variable “dato”.

Una vez el valor de "dato" está comprendido en la lista aceptada, el programa sale del bucle y pasa por una serie de condicionales "IF". Cada uno de estos condicionales compara que "dato" sea igual a uno de los valores asignados a las órdenes, y el programa solo entra en el "IF" donde ambos valores coinciden,

```
if (dato == 49){

  delayTime = delayMax;
  Serial.println("RefOk");
  gotoStart();
  delay(25);
  Serial.println("R");
  dato = 0;

}
```

Imagen 75: Código orden marcha a referencia

El mensaje "1", valor 49 en la tabla ASCII, es la orden de marcha a referencia. Se reinicia, como se va a hacer en el resto de las órdenes, el valor de "delayTime" al valor de menor frecuencia, se envía el mensaje que confirma la recepción de la orden, "RefOk", y se utiliza la función "gotoStart". Una vez el eje llega a referencia se envía el mensaje "R", que confirma la llegada a referencia al programa principal.

El mensaje "2", valor 50 en la tabla ASCII, es la orden de bajada en busca de precinto.

En esta orden el eje Z va a bajar paso a paso comprobando el valor de la célula de carga para parar cuando haya contacto firme.

Se utilizan 3 variables, aparte de las de control de "step", en esta tarea: "j", "senforce" y "BOT_END".

"senforce" va a contener la diferencia entre el valor recogido por el sensor de fuerza y la tara que se realiza sin carga en la posición de referencia. Se inicializa "senforce" a 1 000 porque el valor resultante del cálculo cuando hay contacto es

```
if (dato == 50){

  delayTime = delayMax;
  Serial.println("DownBOk");
  BOT_END = 0;
  senforce = 1000;
  j = -4000;
  digitalWrite(IMANP, HIGH);
  while(senforce > -100000){
    j++;
    if(j > 200){
      senforce = hx711.read()-tara;
      senforce = senforce;
      j = 0;
    }else{
      senforce = 1000;
    }

    step(BAJAR, X_DIR, X_STP);
    BOT_END = digitalRead(X_END_BOT);
    if(BOT_END > 0){
      senforce = -101000;
    }
    if(delayTime>=(delayMin)){
      delayTime = delayTime - 0.5;
    }
  }
  delay(25);
  Serial.println("FCDown");
  dato = 0;
}
```

Imagen 76: Código orden bajada en busca de precinto.

inferior a 0. Los números relacionados con la célula de carga son elevados porque no se ha utilizado un factor de conversión para pasar los valores de salida a gramos, ya que no era necesario.

"BOT_END" va a contener la lectura del final de carrera instalado en la parte inferior del eje. Se inicializa a 0 porque en caso de contacto se produce un "True" en la entrada digital, que es un 1.

"j" va a ser utilizado como temporizador para la toma de medidas con la célula de carga. Esto es necesario porque la toma de medidas en cada paso ralentiza el movimiento del motor por el tiempo de procesado. Se inicializa a – 4000 para que durante parte de la bajada no se realicen mediciones, ya que durante el bucle de bajada se mide cuando "j" vale 200, y cada vez que se mide se cambia el valor de "j" por 0, para hacer un temporizado de 200 pasos entre mediciones. Esto supone que en los primeros 4200 pasos no se toman medidas, lo cual permite al motor bajar sin cambios de velocidad por el procesado.

Al entrar en esta orden se envía el mensaje "DownOK" como confirmación de entrada en la orden, se enciende el electroimán del cabezal, y se entra en el bucle de bajada.

En el bucle de bajada se realizan pasos del motor hasta que el valor de "senforce" sea inferior a -100000. El valor escogido se iteró buscando un valor en el cual la bajada no reaccionase a pequeños toques, solo a presiones altas contra los precintos, que son robustos, para permitir que la punta llegase a precintos que estuviesen relativamente enterrados.

Dentro de este bucle se va a aumentar el valor de "j", y cuando su valor llega a 200, se toma una medición usando la función "read", se le resta la tara, y se guarda en "senforce", antes de modificar el valor de "j" a 0. Mientras "j" es inferior a 200 lo que el bucle hace es cambiar el valor de "senforce" a 1000, un valor que no va a provocar que se termine el bucle.

En el bucle se realiza 1 paso en cada repetición, y se mide también el valor de "BOT_END", para posibles casos donde ocurriese algún error y la condición de parada no se cumpliese antes de llegar al final del recorrido del eje. En estos casos, donde "BOT_END" cambia su valor a 1, se modifica el valor de "senforce" a -101000, que provoca que se termine el bucle.

Al terminar se envía el mensaje “FCDown” para comunicar que se ha bajado el cabezal a la caja de precintos.

```
if (dato == 51){

  delayTime = delayMax;
  Serial.println("UpOk");

  npiezas = 1;
  TOP_END = 0;
  senforce = 50;
  j = -6000;
  i = 0;
  while(senforce < 200000){
    j++;
    if((j == 200)and(i<5)){
      senforce = hx711.read()-tara;
      senforce = senforce;
      j = 0;
      i++;
    }else{
      senforce = 50000;
    }
    if(senforce < 20000){
      npiezas = 0;
      digitalWrite(IMANP, LOW);
      senforce = 300000;
    }
    if((senforce > 20000)and(senforce < 80000)){
      npiezas = 1;
    }
    if(senforce > 80000){
      npiezas = 2;
      digitalWrite(IMANP, LOW);
      senforce = 300000;
    }

    step(SUBIR, X_DIR, X_STP);
    TOP_END = digitalRead(X_END_TOP);
    if(TOP_END > 0){
      senforce = 300000;
    }
    if(delayTime>=(delayMin)){
      delayTime = delayTime - 0.5;
    }
  }
  delay(25);
  if(npiezas == 1){
    Serial.println("B");
  }else{
    delayTime=delayMax;
    gotoStart();
    Serial.println("M");
  }
  dato = 0;
}
```

Imagen 77: Código orden subida con precinto

El mensaje “3”, valor 51 en la tabla ASCII, es la orden de subida con precinto.

Esta orden tiene dos partes: subida antes de revisar la carga, y subida después de revisar la carga.

Se utiliza la variable “j” como temporizador una vez más, de la misma manera que en la orden de bajada en busca de precinto. Se inicializa a -6000 para dar un margen amplio de subida antes de tomar la medida, y que no haya contacto entre la carga y el contenido de la caja de precintos.

Después de enviar el mensaje de confirmación de entrada en la orden, “UpOk”, se entra en un bucle que se va a repetir mientras el valor guardado en la variable “senforce” sea

menor que 200000. El valor de "senforce" en este caso será positivo por haber peso extra colgado del cabezal, por el precinto enganchado.

En este bucle primero se comprueba el valor de "j" y de la variable "i", inicializada a 0, que actuará de contador. En el caso de que el valor de "j" sea 200, y el de "i" sea menor que 5, se toma una medida con la célula de carga. Cuando se toma el valor de la célula de carga se aumenta la cuenta de "i" en 1 y se cambia el valor de "j" a 200. Se usa "i" para repetir este procedimiento 5 veces, para evitar posibles falsos positivos.

En caso de que no se cumpla alguna de las dos condiciones se modifica el valor de "senforce" a 50000, que es un valor que no provoca que se salga del bucle, que además coincide con el valor medio de "senforce" cuando se carga 1 precinto.

Después de tomar el valor de "senforce" se toman acciones dependiendo de su valor.

En caso de que su valor se encuentre entre 20000 y 80000 se cambia el valor de la variable "npiezas" a 1, y se continua con las acciones del bucle al ser el valor de "senforce" menor de 200000.

En caso de que el valor de "senforce" sea menor de 20000, se considera que no hay precinto cargado, se cambia el valor de la variable "npiezas" a 0, se apaga el electroimán del cabezal, y se modifica el valor de "senforce" a 300000, lo cual sacará el programa del bucle al terminar la iteración.

En caso de que el valor de "senforce" sea mayor de 80000, se considera que hay más de 1 precinto cargado, se cambia el valor de la variable "npiezas" a 2, se apaga el electroimán del cabezal, para dejar caer los precintos, y se modifica el valor de "senforce" a 300000, lo cual sacará el programa del bucle al terminar la iteración.

Después de tomar acciones dependiendo del valor de "senforce" se hace el paso del motor, y se comprueba el valor de "TOP_END" para evitar llegar al final de eje. En caso de que el valor de "TOP_END" sea 1 se cambia el valor de "senforce" a 300000 para terminar el bucle.

El movimiento de subida hasta este punto de la orden continúa hasta diferentes posiciones del eje dependiendo de las mediciones. En el caso de que se recoja 1 precinto se sube hasta el final del eje, pero en caso de que la cantidad de precintos sea distinta el movimiento termina en el punto en el que se mide.

Al salir del bucle, si la cantidad de precintos medida es distinta a 1, se utiliza la función "gotoStart" para terminar de subir el cabezal. Al mismo tiempo en que se comprueba el valor de "npiezas" para determinar si es necesario seguir subiendo, se decide que mensaje enviar de vuelta al programa principal. Se envía "B" para los casos en los que se ha extraído 1 precinto, y "M" para los otros casos.

Un último detalle es que en el caso de que se corte el movimiento por detectar una cantidad de precintos distinta a 1, se reinicia el valor de "delayTime" al valor máximo para evitar posibles saltos de paso por la pérdida de velocidad, causada por el tiempo de procesado.

```
if (dato == 52){

  delayTime = delayMax;
  Serial.println("DownDOk");
  BOT_END = 0;
  for (int i = 0; i < 10000; i++){
    step(BAJAR, X_DIR, X_STP);
    BOT_END = digitalRead(X_END_BOT);
    if(BOT_END > 0){
      i = 10000;
    }
    if(delayTime>=delayMin){
      delayTime = delayTime - 0.5;
    }
  }

  digitalWrite(IMANOIZQ, LOW);
  delay(100);
  digitalWrite(IMANP, LOW);
  delay(500);
  digitalWrite(IMANOIZQ, HIGH);
  Serial.println("D");
  dato = 0;
}
```

Imagen 78: Código orden de descarga

El mensaje "4", valor 52 en la tabla ASCII, es la orden de descarga.

En esta orden un bucle "FOR" va a bajar 10 000 pasos del motor, para bajar el precinto colgado del cabezal al bloque de captura.

Primero se envía el mensaje "DownOK" al programa principal como confirmación de entrada en la orden.

Después el bucle "FOR" mueve el motor 1 paso 10 000 veces. Esta cifra se obtuvo probando que cantidad de pasos eran necesarios para que el precinto acabase en la posición deseada.

Durante la bajada se comprueba en cada paso el valor de "BOT_END", y en caso de que el final de carrera inferior sea pulsado se cambia el valor de la variable "i", el contador del bucle "FOR", a 10000, terminando la bajada del eje.

Con el eje bajado hasta la posición de descarga, se apaga el electroimán del cabezal para dejar caer el precinto, pero primero, para asegurar que la trampilla se mantiene cerrada tras el impacto del precinto, se enciende el solenoide izquierdo de la trampilla. Esto hace que la trampilla se mantenga sujeta durante el impacto, ya que en caso de estar suelta la vibración puede mover la trampilla, y permitir que el precinto acabe inclinado. Después de una espera de 500 milisegundos se apaga el solenoide izquierdo, que es tiempo suficiente para que el precinto caiga desde el cabezal y entre en reposo.

```
if (dato == 53){

  dato = 0;
  digitalWrite(IMANP, LOW);
  Serial.println("EB");
  while((digitalRead(BOTON) == 0)and(dato != 57)){
    delay(50);
    dato = Serial.read();
  }

  if(dato != 57){
    delay(100);
    j = 0;
    while(j < 50){
      tara = hx711.read();
      taramed = taramed + tara;
      j++;
    }
    tara = taramed/50;
    taramed = 0;
    Serial.println("P");
    ProgOn=1;
    Serial.println("WBO");
  }

  if(dato == 57){
    Serial.println("OL");
  }

  dato = 0;

}
```

Imagen 79: Código orden espera de solicitud de precinto

El mensaje "5", valor 53 en la tabla ASCII, es la orden espera de solicitud de precinto.

Esta orden tiene dos partes. En la primera un bucle "WHILE" lee el buffer del puerto serie y comprueba el valor del botón de solicitud, a la espera de una de dos cosas: la solicitud del precinto o la orden de salir de la espera. En la segunda parte se hace una tara para obtener el valor de la medición de la célula de carga en reposo.

Antes de entrar en el bucle de espera se hacen dos cosas: primero se apaga el electroimán del cabezal, para que cuando se hace un apagado de emergencia del robot que se haya quedado encendido no sea un problema, y después se envía el mensaje "EB" para confirmar que se ha entrado en la espera de solicitud de precinto.

El bucle comprueba dos cosas, y se ejecuta mientras no se cumplan ambas: que la lectura del botón de solicitud sea o, no pulsado, y la lectura del puerto serie no contenga el valor 57 de la tabla ASCII, que es el "9". El bucle actúa solo como una espera, y en cuanto alguna de las dos condiciones no se cumple se pasa a la segunda parte.

En la segunda parte, dependiendo de cuál sea el motivo por el cual se haya salido del bucle de espera se toman unas acciones u otras.

En el caso de que se reciba el mensaje "9" se envía por la comunicación serie el mensaje "OL" y se sale de la orden.

En el caso de que se pulse el botón de solicitud de precinto se ejecuta un bucle que toma la medida de la célula de carga 50 veces, usando la variable "j" como contador, y hace la media sumando todos los valores y dividiendo entre 50. El valor resultado se guarda en la variable "tara". Una vez hecha la tara se envían dos mensajes, "P" y WBO", que confirman el completado de la tara y de la orden.

```
if (dato == 54){

  Serial.println("LuzOnOk");
  digitalWrite(LUZ, LOW);
  delay(500);
  Serial.println("LuzOnDone");
  dato = 0;

}
```

Imagen 80: Código orden encender iluminación

El mensaje "6", valor 54 en la tabla ASCII, es la orden encender iluminación.

En esta orden se enciende el LED del cabezal, para tomar la foto. Se envía el mensaje "LuzOnOk" para confirmar la entrada en la orden, se enciende el LED, y antes de enviar el mensaje para confirmar el fin de la orden se hace una espera de 500 milisegundos para dar tiempo a LED a calentarse, ya que sin esta espera la secuencia del programa principal envía la orden de tomar foto demasiado rápido y la foto se toma sin iluminación.

```
if (dato == 55){

  Serial.println("LuzOffOk");
  digitalWrite(LUZ, HIGH);
  delay(25);
  Serial.println("LuzOffDone");
  dato = 0;

}
```

Imagen 81: Código orden apagar iluminación

El mensaje "7", valor 55 en la tabla ASCII, es la orden apagar iluminación.

En esta orden se apaga el LED. Primero se envía el mensaje "LuzOffOK" para confirmar la entrada en la orden, y después de apagar el LED se envía el mensaje "LuzOffDone" para confirmar que se ha completado la orden.

```
if (dato == 56){

  Serial.println("TapaOk");
  digitalWrite(IMANODER, LOW);
  delay(200);
  digitalWrite(IMANODER, HIGH);
  delay(800);
  Serial.println("TapaDone");
  digitalWrite(IMANOIZQ, LOW);
  delay(200);
  digitalWrite(IMANOIZQ, HIGH);
  dato = 0;

}
```

Imagen 82: Código orden expulsar precinto

El mensaje "8", valor 56 en la tabla ASCII, es la orden expulsar precinto.

En esta orden se deja caer el precinto del bloque de captura. Para ello se ejecuta una secuencia de encendidos y apagados de los solenoides de la trampilla que la abren para liberar el precinto y luego la cierran.

Después de enviar el mensaje de confirmación de entrada en la orden, “TapaOk”, se ejecuta la siguiente secuencia: encender solenoide derecho, apagar solenoide derecho, encender solenoide izquierdo, apagar solenoide izquierdo.

La trampilla se encuentra cerrada cuando está a la izquierda. Con ambos solenoides apagados, se enciende el derecho para cambiar de posición la trampilla a la posición de abierto. Se hace una espera de 200 milisegundos para dar tiempo al movimiento a completarse y se apaga el solenoide, con el movimiento completado.

Se hace una espera más larga, de 800 milisegundos, para que el precinto tenga tiempo de caer por el hueco de la trampilla, y se envía el mensaje “TapaDone” al programa principal, que confirma que se ha liberado el precinto. El mensaje no se envía al final de la orden como en el resto porque lo que queda de secuencia no tiene efecto en las acciones del programa principal, con lo que se puede realizar en segundo plano.

Después de la espera se enciende el solenoide izquierdo durante 200 milisegundos para volver a cerrar la trampilla, y se apaga.

Programa *Python*

Para este programa se necesitan dos librerías, una de comunicación por TCP-IP y otra que permita tomar las fotos de las cámaras y guardarlas en el directorio designado. La librería escogida para la toma de fotos es OpenCV, una librería muy conocida y fácil de usar, y será también el motivo por el cual este programa está en *Python*, aunque se pueda utilizar en C++, que, como lenguaje, permite procesar más rápido.

La librería de comunicación TCP-IP en Windows para C++ es Winsock, hecha por Microsoft, pero OpenCV y Winsock tienen problemas de compatibilidad, al menos así ha sido en experiencias previas que he tenido tratando de utilizar ambas en el mismo programa, así que siendo que la velocidad de procesado no es un problema en el proyecto el programa se hizo en *Python* para esquivar las incompatibilidades. La librería utilizada para TCP-IP en *Python* será SocketIO.

Para acceder a las cámaras se utiliza la función "VideoCapture" de OpenCV, a la cual solo es necesario introducir el índice de la cámara que se quiere utilizar en la lista de cámaras conectadas al ordenador. Esta función devuelve un objeto de tipo "videocapture", que otras funciones de OpenCV utilizan. El objeto devuelto por la función se guarda en una variable distinta para cada cámara que se utiliza, en el caso de este programa "cam1" para la cámara de la caja de precintos, "cam2" para la cámara izquierda del bloque de captura y "cam3" para la cámara derecha del bloque de captura.

Después se prepara el socket de la comunicación TCP-IP. Este programa va a actuar como servidor, y el programa de *Sherlock* será el cliente que busca la comunicación con el servidor.

Se crea el socket, se vincula un puerto al socket y se pone en modo escucha. Cuando el socket está en modo escucha significa que está a la espera de conexiones.

```
# Crear socket TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Vincular el socket al puerto
server_address = ('localhost', 54000)
print(sys.stderr,'starting up on %s port %s' % server_address)
sock.bind(server_address)
# Poner el socket en modo escucha a la espera de conexiones
sock.listen(1)
```

Imagen 83: Código configuración servidor TCP-IP

Con la conexión preparada se realizan dos cosas más: la configuración de los parámetros de las cámaras, y un parche.

```
a = cam1.set(cv2.CAP_PROP_CONTRAST,255)#255
b = cam1.set(cv2.CAP_PROP_EXPOSURE,-9)#-9
c = cam1.set(cv2.CAP_PROP_BRIGHTNESS,0)

d = cam2.set(cv2.CAP_PROP_CONTRAST,33)
#e = cam2.set(cv2.CAP_PROP_EXPOSURE,-8)
f = cam2.set(cv2.CAP_PROP_BRIGHTNESS,-4)

g = cam3.set(cv2.CAP_PROP_CONTRAST,33)
#h = cam3.set(cv2.CAP_PROP_EXPOSURE,-8)
i = cam3.set(cv2.CAP_PROP_BRIGHTNESS,-4)
```

Imagen 84: Código configuración cámaras

En la imagen 84 se ven las modificaciones de los parámetros de las cámaras. Se ajustan 3 parámetros para conseguir las imágenes deseadas: contraste, tiempo de exposición y brillo. Se utiliza la función "set" de OpenCV para hacer estas configuraciones, que tiene como entradas el parámetro que se desea modificar y el nuevo valor para el parámetro. Un detalle relevante es que en las 3 webcam utilizadas los parámetros de la anterior ejecución si se quedan guardados, a diferencia de los motores de los ejes X e Y. Es relevante por el valor del tiempo de exposición de las cámaras 2 y 3, las del bloque de captura. En la imagen 84 se ve que el código que modifica este parámetro en estas cámaras está comentado, y es porque se ha dejado en automático. En el diseño del sistema de visión del bloque de captura se explicó que era necesario dejar en automático el tiempo de exposición de estas cámaras, para poder utilizar su iluminación propia y no necesitar añadir iluminación externa. Como desde OpenCV no hay acceso a poner en automático el ajuste del tiempo de exposición, se modificó desde un programa externo que si tiene acceso y se utilizó el hecho de que la configuración se queda guardada. El código comentado se ha dejado porque tiene el valor al cual el sistema automático suele poner el tiempo de exposición.

```
ret1, frame1 = cam1.read()
if(ret1==True):
    cv2.imwrite('Input1.jpg', frame1)
ret2, frame2 = cam2.read()
if(ret2==True):
    cv2.imwrite('Input2.jpg', frame2)
ret3, frame3 = cam3.read()
if(ret3==True):
    cv2.imwrite('Input3.jpg', frame3)
```

Imagen 85: Código parche primera imagen

El parche que se ve en la imagen 85 consiste en la toma de una imagen por parte de cada una de las cámaras. Esto ha sido necesario añadirlo porque a veces en la primera ejecución, cuando se toma la primera imagen, aparece en negro. Como solo ocurre la

primera vez que se toma una foto después de encender el programa, se añadió este trozo de código para evitar que ocurra cuando es necesario analizar la imagen.

La función "read" recoge el contenido del último *frame* que ha tomado la cámara, pues esta desde que se ejecuta "VideoCapture" está tomando imágenes como si fuese un video, y guarda su contenido en la variable "frame". La variable "ret" guarda si se ha conseguido recoger un *frame*. Después se utiliza la función "imwrite" para guardar este *frame* recogido como una imagen en el sistema.

Después del parche se entra en un bucle infinito que espera a que llegue una conexión al socket que se ha creado, y una vez aparece se entra en otro bucle infinito que espera y recoge aquello que llegue por la conexión establecida. Dependiendo del mensaje que llega, se toma una imagen con una cámara u otra, y se guarda en el directorio definido para que el programa de *Sherlock* pueda leerlas.

```
if data == b'1':

    ret1, frame1 = cam1.read()
    if(ret1==True):
        cv2.imwrite('Input1.jpg', frame1)
    print (sys.stderr, 'sending data back to the client')
    connection.sendall(data)

if data == b'2':

    ret2, frame2 = cam2.read()
    if(ret2==True):
        cv2.imwrite('Input2.jpg', frame2)
    print (sys.stderr, 'sending data back to the client')
    connection.sendall(data)

if data == b'3':

    ret3, frame3 = cam3.read()
    if(ret3==True):
        cv2.imwrite('Input3.jpg', frame3)
    print (sys.stderr, 'sending data back to the client')
    connection.sendall(data)
```

Imagen 86: Código toma imagen según lectura

Conclusiones

El objetivo del proyecto, demostrar que con la idea planteada era posible dispensar precintos y obtener su código, se ha cumplido, y la idea ha demostrado poder ser escalable a recipientes de precintos mayores.

Se ha diseñado un prototipo de laboratorio en el que se han integrado tres ejes lineales, un cabezal y dos sistemas de visión para la recogida y lectura de precintos.

Se han probado diferentes tipos de sensores para las distintas funciones del prototipo con los que se han obtenido los resultados necesarios para el proyecto. Como trabajo futuro se plantea el uso de sensores industriales.

Se ha programado la secuencia de control para el prototipo que toma las imágenes, localiza los precintos y recoge un precinto para una descarga controlada.

Se ha diseñado un algoritmo que compensa los puntos débiles del sistema de visión 2D mediante movimiento, permitiendo al sistema, de base simple, que extraiga información donde normalmente sería necesario el uso de un sistema de visión 3D.

El cabezal diseñado da buenos resultados y realiza sin problemas la función básica para la que se desarrolló. Como trabajo futuro se plantea el rediseño del cabezal para poder además acceder a los extremos de la caja.

Presupuesto

Tabla 1: Coste de los materiales

Material	Modelo	Cantidad	Precio (€)	Total (€)
Licencias	Sherlock (prueba)	1	0	0
	Arduino IDE	1	0	0
	Visual Studio Community 2019	1	0	0
	Windows 10 Home	1	145	145
Ordenador		1	500	500
Microcontrolador	Arduino Uno	1	20	20
TOTAL				665

Tabla 2: Coste de la mano de obra

Concepto	Descripción	Cantidad (h)	Precio (€/h)	Total (€)
Estudio	Estudio y selección de componentes	60	30	1800
Desarrollo	Desarrollo del software	120	30	3600
Pruebas	Pruebas del software en el prototipo	20	30	600
TOTAL				6000

Coste total del proyecto:	6.665 €

Anexos

1. Variables programa Sherlock

	Name	Value
P	PosicionMetal	
N	PosX	
N	PosY	
S	EMmsj	SRF+\r
S	AMmsj	SRF-\r
S	HEPmsj	SIP:19C:0001!\r
S	VBRmsj	SIP:56E:0059!\r
S	PBRmsj	SIPL:574:00000000!\r
S	DBRmsj	SIP:570:0028!\r
S	BRmsj	REFGO!\r
S	CMP0msj	SIP:5A0:0022!\r
S	CMP1msj	SIP:5D0:0022!\r
S	CMP2msj	SIP:600:0022!\r
S	CVP0Xmsj	SIP:59C:003C!\r
S	CVP0Ymsj	SIP:59C:00C8!\r
S	CVP1Xmsj	SIP:5CC:003C!\r
S	CVP1Ymsj	SIP:5CC:00C8!\r
S	CVP2Xmsj	SIP:604:003C!\r
S	CVP2Ymsj	SIP:604:00C8!\r
S	CMRRmsj	SIPL:560:001E0000!\r
S	IPmsj	SPGO!\r
N	timeout	100.00
N	IndX	
N	IndY	
S	CPP0Xmsj	
S	CPP0Ymsj	
S	CPP0Xent	
S	CPP0Xdec	
S	CPP0Yent	
S	CPP0Ydec	
S	MsjPart1	SIPL:590:
S	MsjPart2	!\r
N	i	
P[]	VectorPosROI	((322.00, 0.00), ...) [2]
P	CoorFinROI	
P	CoorIniROI	
P[]	CentVect	
N	LCentVect	
S	POSOKmsjX	
S	POSOKmsjY	
S	POSOKcomp	POSOK
N	Variable	0.00
S	MsjBton	
S	CompBton	P
B	BPress	
B	varA	False
S	varB	res
N	PEjec	0.00
S	MsgCompNPiezas	
S	CompPB	B
N	MCL	
N	NPuertoMX	0.00
N	NPuertoMY	2.00
N	NPuertoArd	3.00
S	CompZRef	R
S	ZRefMsj	
N	MRL	
P[]	Punto1-1	((322.00, 0.00), ...) [2]
P[]	Punto1-2	((148.00, 0.00), ...) [2]
P[]	Punto1-3	((0.00, 0.00), ...) [2]
P[]	Punto2-1	((322.00, 0.00), ...) [2]
P[]	Punto2-2	((148.00, 0.00), ...) [2]
P[]	Punto2-3	((0.00, 0.00), ...) [2]
N	NPuntosRecorridos	
S	MsjP1-Y	SIPL:5C0:00000000!\r
S	MsjP2-Y	SIPL:5C0:00040000!\r
S	MsjPX-1	SIPL:5C0:00010000!\r
S	MsjPX-2	SIPL:5C0:00090000!\r
S	MsjPX-3	SIPL:5C0:000F0000!\r
N	PCalcX	
N	PCalcY	
S	AddCod	1
S	Cam2String	
S	Cam3String	
N	NIntentos	

2. Código Arduino

```
#define EN  8
#define X_DIR 5
#define X_STP 3
#define X_END_TOP 6
#define X_END_BOT 10
#define SUBIR 1
#define BAJAR 0
#define IMANP 2
#define LUZ 7
#define IMANOIZQ 9
#define IMANODER 11
#define BOTON 4
#include <SoftwareSerial.h>
#include <Q2HX711.h>



float delayTime = 0;        //Delay between each pause (uS)
int delayMax = 700;
int delayMin = 400;
int direccion;
int dato, valorFuerza = 0,TOP_END = 0, BOT_END = 0, BajarEJE;
int npiezas = 0;
int ProgOn = 1;
long int valorStep = 0;
int alg = 1, pieza = 0;
int i = 0, j = 0;
long senforce = 0;
long tara = 0, taramed = 0;

int flexiForcePin = A0; //analog pin 0

void step(boolean dir, byte dirPin, byte stepperPin)

{
 digitalWrite(dirPin, dir);
 digitalWrite(stepperPin, HIGH);
 delayMicroseconds((int)delayTime);
 digitalWrite(stepperPin, LOW);
 delayMicroseconds((int)delayTime);

 if(dir==1)
 {
  valorStep=valorStep+1;      //CONTADOR PARA SABER EL NÚMERO DE PASOS
 }
 else
 {
  valorStep=valorStep-1;
 }
}
```

```
void gotoStart()
{
 TOP_END=0;
 while (TOP_END==0)
 {
  step(SUBIR, X_DIR, X_STP);          //LLEVAR EL EJE EN LA DIRECCION NECESARIA PARA
SUBIRLO CAMBIANDO FALSE POR TRUE O AL REVES
  TOP_END = digitalRead(X_END_TOP);
  if(delayTime>=delayMin){
   delayTime = delayTime - 0.5;
  }
 }
 valorStep=0;
}

Q2HX711 hx711(A2, A3);

void setup(){

 Serial.begin(9600);
 Serial.print("Conexión Establecida.");

 pinMode(13, OUTPUT);
 pinMode(X_DIR, OUTPUT);
 pinMode(X_STP, OUTPUT);
 pinMode(X_END_TOP,INPUT);        //PIN PARA EL FINAL DE CARRERA SUPERIOR
 pinMode(13,OUTPUT);              //LED PARA COMPROBAR EL ESTADO
 pinMode(IMANP, OUTPUT);
 pinMode(LUZ, OUTPUT);
 pinMode(IMANOIZQ, OUTPUT);
 pinMode(IMANODER, OUTPUT);
 pinMode(BOTON,INPUT);
 pinMode(EN, OUTPUT);

 digitalWrite(EN, LOW);
 digitalWrite(LUZ, HIGH);

//-------------------------------------------------------------------------------------------------------------------

 digitalWrite(IMANODER, HIGH);
 digitalWrite(IMANOIZQ, LOW);
 delay(200);
 digitalWrite(IMANOIZQ, HIGH);

//-------------------------------------------------------------------------------------------------------------------

}

void loop(){

  dato = 0;
```

```
  while((dato < 49)or(dato > 56)){
   dato=Serial.read();          //ESPERA A RECIBIR LA PRIMERA CONFIRMACION DE SHERLOCK
   delay(200);
  }

//-------------------------------------------------------------------------------------------------------------------

  if (dato == 49){                                                    //1 -> Ir a referencia

   delayTime = delayMax;
   Serial.println("RefOk");
   gotoStart();
   delay(25);
   Serial.println("R");
   dato = 0;

  }

//-------------------------------------------------------------------------------------------------------------------

  if (dato == 50){                                  //2 -> Bajar electroimanP (busqueda de pieza)

   delayTime = delayMax;
   Serial.println("DownBOk");
   BOT_END = 0;
   senforce = 1000;
   j = -4000;
   digitalWrite(IMANP, HIGH);
   while(senforce > -100000){
    j++;
    if(j > 200){
     senforce = hx711.read()-tara;
     senforce = senforce;
     j = 0;
    }else{
     senforce = 1000;
    }
    step(BAJAR, X_DIR, X_STP);
    BOT_END = digitalRead(X_END_BOT);
    if(BOT_END > 0){
     senforce = -101000;
    }
    if(delayTime>=(delayMin)){
     delayTime = delayTime - 0.5;
    }
   }
   delay(25);
   Serial.println("FCDown");
   dato = 0;
  }

//-------------------------------------------------------------------------------------------------------------------
```

```
if (dato == 51){                                    //3 -> Subir electroimanP (final de carrera)

 delayTime = delayMax;
 Serial.println("UpOk");

 npiezas = 1;
 TOP_END = 0;
 senforce = 50;
 j = -6000;
 i = 0;
 while(senforce < 200000){
  j++;
  if((j == 200)and(i<5)){
    senforce = hx711.read()-tara;
    senforce = senforce;
    j = 0;
    i++;
  }else{
    senforce = 50000;
  }
  if(senforce < 20000){
    npiezas = 0;
    digitalWrite(IMANP, LOW);
    senforce = 300000;
  }
  if((senforce > 20000)and(senforce < 80000)){
    npiezas = 1;
  }
  if(senforce > 80000){
    npiezas = 2;
    digitalWrite(IMANP, LOW);
    senforce = 300000;
  }

  step(SUBIR, X_DIR, X_STP);
  TOP_END = digitalRead(X_END_TOP);
  if(TOP_END > 0){
    senforce = 300000;
  }
  if(delayTime>=(delayMin)){
    delayTime = delayTime - 0.5;
  }
 }
 delay(25);
 if(npiezas == 1){
  Serial.println("B");
 }else{
  delayTime=delayMax;
  gotoStart();
  Serial.println("M");
 }
```

```
  dato = 0;
 }

//-------------------------------------------------------------------------------------------------------------------

 if (dato == 52){                                        //4 -> Bajar electroimanP (descarga)

  delayTime = delayMax;
  Serial.println("DownDOk");
  BOT_END = 0;
  for (int i = 0; i < 10000; i++){
   step(BAJAR, X_DIR, X_STP);
   BOT_END = digitalRead(X_END_BOT);
   if(BOT_END > 0){
    i = 10000;
   }
   if(delayTime>=delayMin){
    delayTime = delayTime - 0.5;
   }
  }

  digitalWrite(IMANOIZQ, LOW);
  delay(100);
  digitalWrite(IMANP, LOW);
  delay(500);
  digitalWrite(IMANOIZQ, HIGH);
  Serial.println("D");
  dato = 0;
 }

//-------------------------------------------------------------------------------------------------------------------

 if (dato == 53){                                        //5 -> Esperar boton

  dato = 0;
  digitalWrite(IMANP, LOW);
  Serial.println("EB");
  while((digitalRead(BOTON) == 0)and(dato != 57)){
   delay(50);
   dato = Serial.read();
  }
  if(dato != 57){
   delay(100);
   j = 0;
   while(j < 50){
    tara = hx711.read();
    taramed = taramed + tara;
    j++;
   }
   tara = taramed/50;
   taramed = 0;
   Serial.println("P");
```

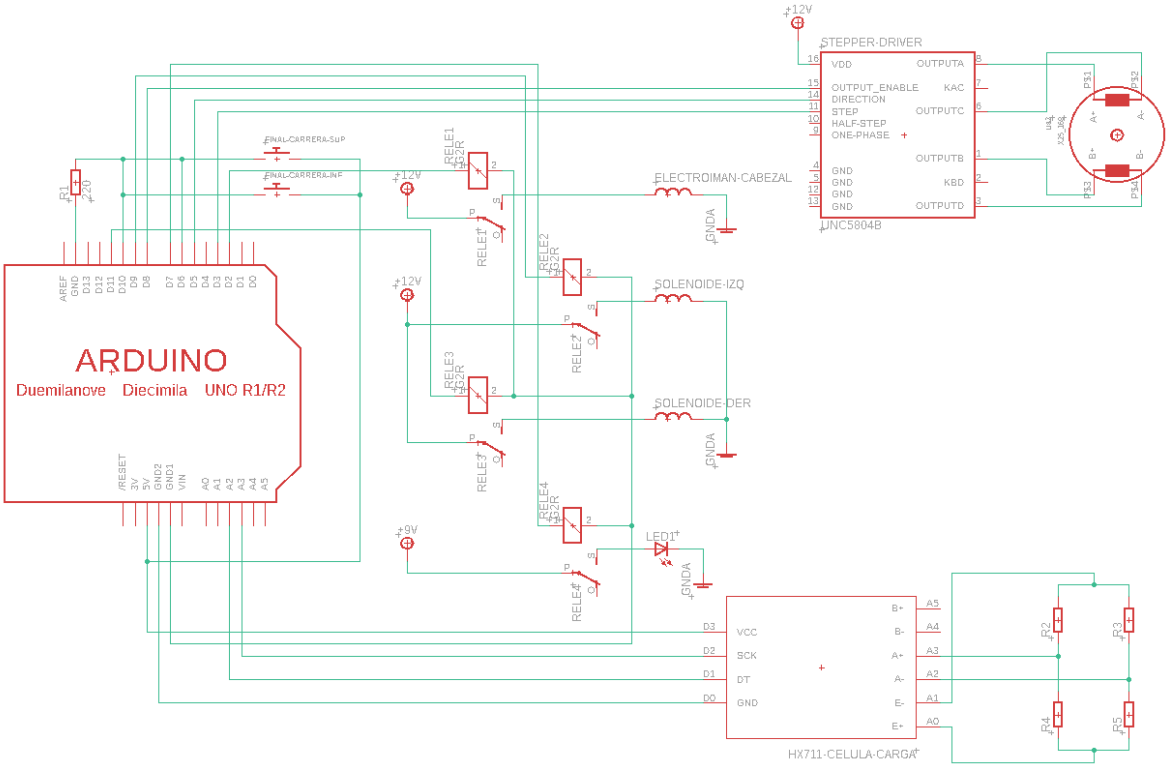
```
      ProgOn=1;
      Serial.println("WBO");
     }
     if(dato == 57){
      Serial.println("OL");
     }
     dato = 0;
    }

//-------------------------------------------------------------------------------------------------------------------

    if (dato == 54){                                                    //6 -> Encender luz

     Serial.println("LuzOnOk");
     digitalWrite(LUZ, LOW);
     delay(500);
     Serial.println("LuzOnDone");
     dato = 0;

    }
//-------------------------------------------------------------------------------------------------------------------

    if (dato == 55){                                                    //7 -> Apagar luz

     Serial.println("LuzOffOk");
     digitalWrite(LUZ, HIGH);
     delay(25);
     Serial.println("LuzOffDone");
     dato = 0;

    }
//-------------------------------------------------------------------------------------------------------------------

    if (dato == 56){                                                  //8 -> ElectroimánO tapa OCR precinto

     Serial.println("TapaOk");
     digitalWrite(IMANODER, LOW);
     delay(200);
     digitalWrite(IMANODER, HIGH);
     delay(800);
     Serial.println("TapaDone");
     digitalWrite(IMANOIZQ, LOW);
     delay(200);
     digitalWrite(IMANOIZQ, HIGH);
     dato = 0;

    }
//-------------------------------------------------------------------------------------------------------------------

}
```

3. Código Python

```
import socket
import sys
import cv2
import os

cam1 = cv2.VideoCapture(2)
cam2 = cv2.VideoCapture(1)
cam3 = cv2.VideoCapture(0)

if not cam1.isOpened():
 raise IOError("Cannot open webcam 1")
if not cam2.isOpened():
 raise IOError("Cannot open webcam 2")
if not cam3.isOpened():
 raise IOError("Cannot open webcam 3")

directory = r'C:\Users\DADIALO\Desktop\PRECINTOS'
os.chdir(directory)


# Crear socket TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Vincular el socket al puerto
server_address = ('localhost', 54000)
print(sys.stderr,'starting up on %s port %s' % server_address)
sock.bind(server_address)
# Poner el socket en modo escucha a la espera de conexiones
sock.listen(1)

a = cam1.set(cv2.CAP_PROP_CONTRAST,255)#255
b = cam1.set(cv2.CAP_PROP_EXPOSURE,-9)#-9
c = cam1.set(cv2.CAP_PROP_BRIGHTNESS,0)

d = cam2.set(cv2.CAP_PROP_CONTRAST,33)
#e = cam2.set(cv2.CAP_PROP_EXPOSURE,-8)
f = cam2.set(cv2.CAP_PROP_BRIGHTNESS,-4)

g = cam3.set(cv2.CAP_PROP_CONTRAST,33)
#h = cam3.set(cv2.CAP_PROP_EXPOSURE,-8)
i = cam3.set(cv2.CAP_PROP_BRIGHTNESS,-4)

ret1, frame1 = cam1.read()
if(ret1==True):
    cv2.imwrite('Input1.jpg', frame1)
ret2, frame2 = cam2.read()
if(ret2==True):
    cv2.imwrite('Input2.jpg', frame2)
ret3, frame3 = cam3.read()
if(ret3==True):
    cv2.imwrite('Input3.jpg', frame3)

while True:
    # Wait for a connection
    print(sys.stderr, 'waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print (sys.stderr, 'connection from', client_address)

        # Receive the data in small chunks and retransmit it
        while True:
```

```
            data = connection.recv(16)
            print (sys.stderr, 'received "%s"' % data)

            if data == b'1':

                ret1, frame1 = cam1.read()
                if(ret1==True):
                    cv2.imwrite('Input1.jpg', frame1)
                print (sys.stderr, 'sending data back to the client')
                connection.sendall(data)

            if data == b'2':

                ret2, frame2 = cam2.read()
                if(ret2==True):
                    cv2.imwrite('Input2.jpg', frame2)
                print (sys.stderr, 'sending data back to the client')
                connection.sendall(data)

            if data == b'3':

                ret3, frame3 = cam3.read()
                if(ret3==True):
                    cv2.imwrite('Input3.jpg', frame3)
                print (sys.stderr, 'sending data back to the client')
                connection.sendall(data)

            else:
                print (sys.stderr, 'no more data from', client_address)
                break

    finally:
        # Clean up the connection
        connection.close()
cam1.release()
cam2.release()
cam3.release()
cv2.destroyAllWindows()
```

4. Esquema conexiones Arduino



Bibliografía

[1] ISO, norma ISO 17712:2013, "Freight containers – Mechanical seals" [En línea] [Consulta: 9 de febrero 2021]. Disponible: https://www.iso.org/standard/62464.html

[2]RFL cargo, "El sello o precinto de los contenedores" [En línea] [Consulta: 9 de agosto 2019]. Disponible: https://rflcargo.com/precinto-contenedor/

[3] Valenciaport, "Valenciaport presenta un nuevo servicio de gestión de precinto que permitirá reducir tiempo y costes y dotará de un mayor control de stock a las navieras", [En línea] [Consulta: 9 de febrero 2021]. Disponible: https://www.valenciaport.com/valenciaport-presenta-un-nuevo-servicio-de-gestion-de-precintos-que-permitira-reducir-tiempos-y-costes-y-dotara-de-un-mayor-control-de-stock-a-las-navieras/

[4] Revista digital Cedex, "Clasificación de los diez primeros smart ports en el sistema portuario español desde una perspectiva económica, social, institucional, medioambiental y el grado de digitalización" [En línea] [Consulta: 9 de febrero 2021]. Disponible: http://ingenieriacivil.cedex.es/index.php/ingenieria-civil/article/view/2415

[5] El Mercantil, "Valencia desarrolla junto a Cosco y los transportistas de ELTC un gestor de precintos", [En línea] [Consulta: 9 de febrero 2021]. Disponible: https://elmercantil.com/2019/12/11/valencia-desarrolla-junto-a-cosco-y-los-transportistas-de-eltc-un-gestor-de-precintos/

[6] Facebook, Transcont Valencia, mensaje del 10 de junio de 2019 sobre la reunión del consejo de calidad del puerto de Valencia, [En línea] [Consulta: 9 de febrero 2021]. Disponible: https://www.facebook.com/transcontvalencia/posts/2316661148545631/

[7] SpanishPorts, "La optimización de los procesos en el puerto de Valencia, objetivo de la Marca de Garantía" [En línea] [Consulta: 9 de febrero 2021]. Disponible: http://www.spanishports.es/texto-diario/mostrar/1267175/optimizacion-procesos-puerto-valencia-objetivo-marca-garantia